MC0122–00

68xxx v9.0

InterTools C Compiler/Assembler Getting Started Manual

TASKING

Quality Development Tools Worldwide

A publication of
TASKING
Documentation Department
Copyright © 1997 TASKING, Inc.

All rights reserved. Reproduction in whole or part is prohibited without the written consent of the copyright owner.

The following trademarks are acknowledged:

DOS16M and DOS/4GW are trademarks of Rational Systems, Inc.
FLEXlm is a registered trademark of Globetrotter Software, Inc.
HP and HP–UX are trademarks of Hewlett–Packard Co.
IBM is a trademark of International Business Machines Corp.
Motorola is a trademark of Motorola, Inc.
MS–DOS and Windows are registered trademarks of Microsoft Corporation.
Solaris is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.
VAX and VMS are trademarks of Digital Equipment Corporation.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

E–mail: support@tasking.com
WWW: http://www.tasking.com

The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, TASKING assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

TASKING reserves the right to change specifications embodied in this document without prior notice.

Table of Contents

CHAPTER 1 1–1

Introduction 1–1

1.1 InterTools Overview 1–1

1.2 InterTools Documentation 1–2

1.2.1 The InterTools Manuals 1–4

1.2.2 Documentation Conventions 1–5

1.2.2.1 Typewriter Font 1–5

1.2.3 How to Use This Documentation Set 1–9

1.3 The InterTools System 1–9

1.3.1 The Compiler 1–11

1.3.2 The Optimizer 1–11

1.3.3 The Run–time Library 1–12

1.3.4 The Assembler 1–12

1.3.5 Utilities 1–13

1.3.6 The Linking Locator 1–13

1.3.7 The Formatters 1–14

1.3.8 The Librarian 1–15

1.3.9 The Global Symbol Mapper 1–15

1.3.10 The Object Size List Utility 1–15

1.3.11 The Symbol List Utility 1–16

1.3.12 CrossView Pro Debugger 1–16

1.4 Before You Start 1–16

1.4.1 Usage Conventions 1–17

1.4.2 InterTools Versions 1–17

1.4.3 Driver Switches 1–17

1.4.4 Invocation Conventions 1–18

1.4.5 Error Message Output (PC only) 1–19

1.5 Additional Help 1–19

1.5.1 Tutorial 1–19

1.5.2 On–line Help 1–20

1.6 Customer Support 1–22
1.6.1 Eligibility 1–22
1.6.2 If You Have a Problem Using the Software 1–23
1.6.3 TASKING Customer Support 1–27
1.7 Comments, Please 1–29

CHAPTER 2 2–1
Installation Guide 2–1
2.1 Introduction 2–1
2.2 Installation on Unix Hosts 2–1
2.2.1 How to Install the Software 2–1
2.2.2 Components of the Release 2–4
2.2.3 Verifying the Installation 2–6
2.2.4 Installation Notes 2–8
2.3 Installation on PC Hosts 2–9
2.3.1 How to Install the Software 2–9
2.3.2 Release Notes 2–10
2.4 Supplementary Reading 2–11
2.5 More Help 2–11

CHAPTER 3 3–1
InterTools Tutorial 3–1
3.1 Introduction 3–1
3.2 Finding the Programs and Setting Up the Path 3–2
3.2.1 bin Directory 3–4
3.2.2 Assembler Files 3–5
3.2.3 Compiler Files 3–7
3.2.4 C++ Compiler Files 3–9
3.2.5 Utilities 3–10
3.2.6 C++ Utilities 3–10
3.2.7 rtlibs Directory 3–11
3.2.8 examples Directory 3–14
3.3 Invoking the Tools 3–15
3.3.1 Invoking the Tools from EDE 3–15
3.3.2 Invoking the Tools Using Command Line 3–19
3.4 Tutorial Examples 3–20
3.4.1 Example 1 3–21

3.4.2 Example 2 3–23
3.4.3 Example 3 3–31
3.4.4 Example 4 3–37
3.4.5 Example 5 3–43
3.5 Introduction to System Building Concepts 3–47
3.5.1 System Initialization 3–48
3.5.2 A5–Relative vs. Separate Data Addressing 3–49
3.5.3 Linking and Locating 3–50
3.5.4 Linking C and Assembly 3–60
3.6 Tutorial Conclusion 3–64

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual introduces the InterTools toolchain to the new user. This Getting Started manual allows you to start using InterTools right away.

MANUAL STRUCTURE

1. Introduction

Introduces the InterTools documentation conventions and organization. Gives an overview of the InterTools toolchain.

2. Installation Guide

Describes how to install the InterTools C Compiler/Assembler on your system.

3. InterTools Tutorial

Guides you through a brief tutorial to get you started using InterTools.

CHAPTER 1

Introduction

1.1 InterTools Overview

This chapter is an introduction to the InterTools Development Environment. In addition to providing a brief overview of the InterTools toolchain, this chapter also discusses the InterTools documentation set, provides information that helps you start using your InterTools software and gives an overview of InterTools customer support.

TASKING, Inc. has designed and built software development tools that address the specific needs of the embedded systems builder. InterTools is a cross–development system with flexible, modular tools that fit together while not dictating a specific approach to system design.

This introduction consists of a brief summary of the software documentation package, how to use the package, conventions used in the documentation, an explanation of each of the tools in the InterTools cross–development system, and technical information for reference before you start using the tools.

1.2 InterTools Documentation

Three manuals make up the InterTools documentation: the *InterTools Getting Started Manual*, the *InterTools C Compiler/Assembler User's Manual*, and the *InterTools C Compiler/Assembler Reference Manual*. The following tables show the contents of each manual:

InterTools Getting Started Manual
1. Introduction 2. Installation Guide 3. InterTools Tutorial

InterTools C Compiler/Assembler User's Manual

1. Introduction
2. 68000 Family C Compiler
3. 68000 Family Assembler
4. Linking Locator
5. Formatter
6. Other Utilities:
   - Introduction
   - Librarian
   - Global Symbol Mapper
   - Symbol List Utility
   - Object Size List Utility
7. Application Notes:
   - Introduction
   - Downloading
   - Linking C and Assembly
   - Pragma Separate (Option Separate)
   - Building Libraries That Do Not Use A5
   - Position–independent Code
   - Getting the Best Code For Your Application
   - Support for the On–board Peripherals of the 68332, 68340, and 68360
   - InterTools FLEXlm Guide (Unix only)

A. Appendix: C Language Specifications

B. Appendix: Compiler Naming Conventions

C. Appendix: Compiler Run Time Conventions

D. Appendix: Object Module Formats

E. Appendix: Compiler/Assembler Driver

F. Appendix: Supplemental Reading

InterTools C Compiler/Assembler Reference Manual

1. Introduction
2. Run–time Library
3. The Assembly Language Reference
4. Source Program Coding
5. Assembler Directives
6. Macro Operations and Conditional Assembly
7. Structured Control Statements
8. Position–independent Code

A. Appendix: Instruction Set Summary

B. Appendix: Character Set

1.2.1 The InterTools Manuals

The *Introduction* chapter in this manual contains a summary of the InterTools system, and valuable technical information about the software in the *Before You Start* section.

The *Installation Guide* chapter in this manual is primarily for the initial installation of the software on your computer system. It contains information on loading the software, verification procedures on Unix, and miscellaneous installation notes.

The *InterTools Tutorial* chapter in this manual contains sample code and exercises which lead you step–by–step through the powerful features of each software tool. This material is useful for investigating a particular function using the sample code provided in the `tutor` directory.

The *InterTools C Compiler/Assembler User's Manual* includes invocation, options, and usage summaries, along with examples for each of the tools and definitions of special terminology and functions. This manual also contains additional information in the appendices on run time and naming conventions, C language extensions, and object module formats.

The *InterTools C Compiler/Assembler Reference Manual* provides information on the run–time libraries and the information necessary to write programs in assembly language. It contains sections on source program coding, assembler directives, macro operations, structured control statements, and position–independent code, as well as a summary of the target instruction set and character set.

1.2.2 Documentation Conventions

In this documentation set, we use a number of typefaces to demonstrate the syntax of various commands and to show sample text you might type at a terminal or observe in a file. The following is a list of these conventions.

1.2.2.1 Typewriter Font

Used for: user input/screen output

`Typewriter` font is used in examples to show what you might type at a terminal: command names, directives, switches, literal filenames, or any other text which must be typed exactly as shown. It is also used in other examples to show what you might see on a screen or in a printed listing. For example:

```
form [prog.ab] [options]
```

`form` is shown in typewriter font because it must be typed exactly as shown when you run the formatter.

```
#include "stdio.h"
main ()
{
   printf ("hello, world.\n");
}
```

The above example shows the contents of a file that you may have output to the screen.

Italics

Used for: value substitution

Italic type indicates categories of items for which you must substitute appropriate values, such as arguments or hypothetical filenames. For example:

```
-c cfn
```

`-c` is in typewriter font because it must be typed exactly as shown. *cfn* is in italics, indicating that you must substitute an appropriate value, in this case the name of a locator command file, in place of *cfn* when using this option.

We also use *italic* type throughout this documentation set to indicate directory names in explanations.

Square Brackets

Used for: optional items

Optional items in syntax, command, or option expressions appear in square brackets ('[' and ']'). As an example, the syntax of the `-w` assembler option is shown below:

```
-w [n]
```

The *n* is optional and thus is enclosed in square brackets.

Vertical Bar

Used for: indicating that only one of the items in the group may be chosen

A vertical bar separating two or more items indicates that a choice must be made among the items. For example:

```
llink prog.[ol|rmp|ln]
```

The user should select one of the filename extensions shown.

Ellipsis

Used for: indicating that one or more items are allowed

An ellipsis (...) is used after an item to indicate that more than one of that item is allowed. The syntax of the `-I` compiler option is shown below:

```
-I dir1 [dir2...]
```

The `-I` must be typed exactly as shown. A directory name must be substituted for *dir1* and after that there may optionally be any number of additional directory names.

Shaded Boxes

Used for: command line examples

Command line examples appear in shaded boxes, as below:

```
llink prog.o1
```

Bounded Boxes

Used for: listing output

Bounded boxes display listings, as below:

```
January 20, 1996 CROSS-REFERENCE: hello.c PAGE 1

i
   Def : hello.h      2
   Ref : hello.c      5

main
   Def : hello.c      2
```

Note: Due to the page width limitations of this manual, a single line may appear as two or more lines. You should type just one line.

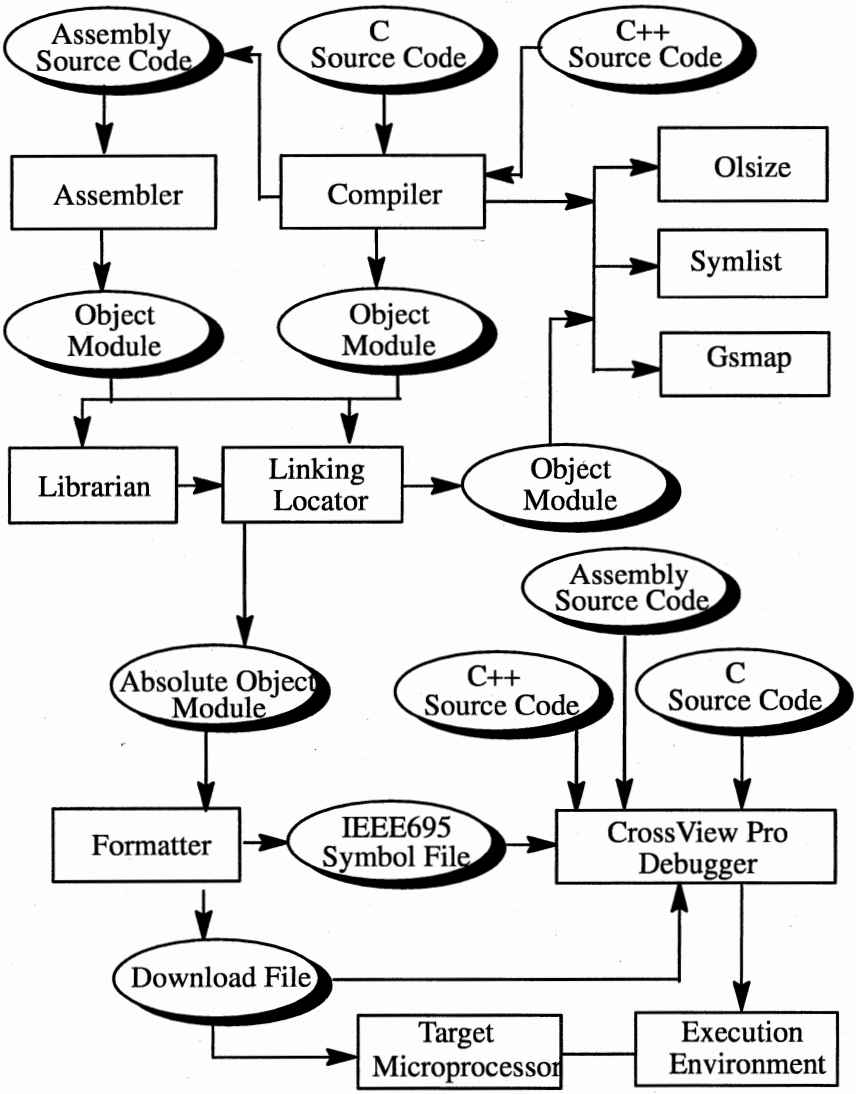
1.2.3 How to Use This Documentation Set

The InterTools documentation is organized to be as flexible and useful as possible. We have considered the varying levels of our users in organizing and writing this manual set. For the inexperienced user of the InterTools system, we have included an *InterTools Tutorial* with exercises and sample code illustrating the basic features of the tools. By "inexperienced users," we mean those who have a working knowledge of the C programming language, assembly language, and the host computer system they are using, but have limited experience using cross compilers, assemblers, and debuggers.

For the experienced user, the *InterTools User's Manual* offers detailed explanations of the tools and their use. By "experienced users," we mean those who are conversant with the processes of compiling, assembling and debugging for embedded systems software applications. Of course, beginning users will find the *InterTools User's Manual* useful for exploring the features of the tools once the *InterTools Tutorial* has been mastered.

1.3 The InterTools System

InterTools is a fully integrated development system. The tools work seamlessly together and share a common object language. The tools produce object language files in ASCII format, while the formatter can produce either binary or ASCII files. Source code and InterTools object modules are portable from host to host, and are interchangeable between all supported hosts. The flexibility of the InterTools solution allows quick and efficient hardware upgrades.


Assembly Source Code
C Source Code
C++ Source Code
Assembler
Compiler
Olsize
Symlist
Gsmap
Object Module
Object Module
Librarian
Linking Locator
Object Module
Assembly Source Code
Absolute Object Module
C++ Source Code
C Source Code
Formatter
IEEE695 Symbol File
CrossView Pro Debugger
Download File
Target Microprocessor
Execution Environment

1.3.1 The Compiler

The compiler translates C source into machine instructions for the target microprocessor. The input is one or more source programs. The C language implemented conforms to the ANSI C standard ANSI/ISO 9899–1990.

Compiler output is an object module suitable for linking with other modules. These object modules can also be catalogued in a library using the librarian utility. The compiler has optional listings which show interleaved source and generated machine instructions, along with cross–reference listings. A pseudo–assembly listing is also available to allow you to view code emitted by the compiler at the assembly language level.

1.3.2 The Optimizer

The InterTools global optimizer improves the speed and reduces the size of the code generated by the InterTools compiler. The global optimizer runs after the compiler front end and before the code generator. It is completely integrated into the compiler system, but its use is entirely optional. Code compiled with the global optimizer can be freely combined with non–optimized code.

The global optimizer performs a variety of optimizations. These optimizations include automatic register allocation, loop optimization, code hoisting, loop rotation, and common subexpression elimination. Please refer to the *C Compiler* chapter in the *InterTools User's Manual* for a complete description of usage and options.

1.3.3 The Run–time Library

The InterTools system includes full run–time libraries: math functions, memory allocation functions, standard I/O functions, string manipulation functions, and floating point routines. No additional license is required to use the library subroutines in your products.

See the *Run–time Library* chapter in the *InterTools Reference Manual* for listings and more detailed information about integrating run–time library modifications.

1.3.4 The Assembler

The InterTools system includes a modern macro assembler. The source format is manufacturer–compatible. That is, existing manufacturer–compatible assembly code is easily reassembled using the InterTools assembler. Minor changes may be needed if the assembled modules are to be invoked by compiled modules. Refer to the *Linking C and Assembly* application note in the *InterTools User's Manual* for more information.

The input to the assembler is one or more source programs. The output is a corresponding number of object modules suitable for linking to other modules. The object modules can be catalogued in a library. Assembler object modules are compatible with C compiler object modules. Source, cross–reference, and symbol table listings are available from the assembler.

1.3.5 Utilities

The InterTools compiler and assembler software includes a full set of utilities. These tools increase programming productivity by reducing the time spent on repetitive software building tasks. A brief description of the utilities is given below. The linking locator and formatter must be run in order to produce a download module. These utilities are described first, followed by optional utilities. Each utility is explained in more detail in the appropriate chapter of the *InterTools User's Manual.*

1.3.6 The Linking Locator

The linking locator integrates the results of separate compilations and assemblies into a single absolute module. This is done in three separate steps, any or all of which can be performed in a single invocation of the linking locator. The first step, called "linking," consists of combining separate object modules into a composite module by resolving references. Usually these object modules are produced by the assembler and/or compiler, but pre–linked object modules may also be used as input. The linking locator searches libraries to satisfy any unresolved references in the module it is constructing.

The second (optional) step, called ROM processing, consists of building initialization segments used to initialize read–write data. All ROM–based systems must execute code to initialize their read–write data, since the initial values cannot be maintained in RAM (random–access memory), and read–write data cannot be allocated in ROM (read only memory). This data could be initialized by large numbers of assignment statements, but it is more convenient and efficient to employ ROM processing instead. Unlike the read–write data, the initialization segment is suitable for placement in ROM. The initial data values are copied from ROM to RAM at the time of initialization by the library routine `rcopy`.

The final step, called locating, consists of assigning absolute target–memory locations to relocatable segments and resolving address references. The linking locator gives you complete control over placement of all code and data, but it also has the capacity to automatically locate collections of segments in bounded areas of the target memory. The output is an object module with absolute addresses substituted where appropriate. A completely located module contains all the information necessary to load and execute the code on the target microprocessor. The linking locator can resolve the problem of storing a program into a fragmented memory space consisting of ROM, RAM, and I/O mapped device addresses.

1.3.7 The Formatters

Form and form695 convert the contents of an absolute object module into one of the industry standard formats, in either an ASCII hex or a binary format. The formats provide for loading of object text, that is, code and data, into the memory of the target processor using a loader. The loader is generally provided by an emulator or other instrumentation system, or by a ROM–resident monitor program. The formatter offers many different formats in order to be compatible with a wide range of loaders.

The input is a module from the linking locator and the output is a formatted load file. The formats may also be used as input to a PROM burner to program read–only memory. See the *Formatter* chapter in the *InterTools User's Manual* for a list of supported formats. The *Object Module Formats* appendix in the *InterTools User's Manual* gives detailed information on each of the supported formats.

1.3.8 The Librarian

The librarian is a tool for managing libraries of program modules at the pre–link or post–link phase of development. The librarian creates, maintains, and selectively lists library index files. A library index file is a text file defining an indexing structure which describes a collection of object modules. It consists of a series of index entries, one for each object module. The librarian's input is taken from the library and/or object modules named on the command line or through options specified on the command line. The object modules named on the command line or in a file are added to the library. Libraries simplify the task of linking modules, since the linking locator can automatically search libraries for required modules.

1.3.9 The Global Symbol Mapper

The global symbol mapper (gsmap) displays global symbols either alphabetically or by address. Gsmap can be used before or after linking or locating to list external names and the definitions of global symbols. The gsmap listing shows an absolute address (after locating), length, class, and alignment for each segment.

1.3.10 The Object Size List Utility

The object size list utility (olsize) lists the total number of words of code, data, and constant data in an object module.

1.3.11 The Symbol List Utility

The symbol list utility (symlist) produces a listing of all global and local symbols. When the debugger option, (-d), is used in compilation or assembly, target locations for source lines of input code are included in the listing. (See the *C Compiler* and *Assembler* chapters in the *InterTools User's Manual* for details on compiler and assembler options.) The input may be any combination of unlinked object modules, linked object modules, and absolute modules. The symlist listing is composed of three parts: a table of executable line numbers and code addresses, a listing of all symbols and their attributes, and an alphabetical list of all symbols with pointers to each symbol's definition and attributes.

1.3.12 CrossView Pro Debugger

The source–level debugger, CrossView Pro provides you with a means of monitoring and controlling execution of the embedded software using the same terms, definitions, and structures found in the original source program. CrossView Pro has complete access to the symbol tables produced by the compiler, and also knows the compiler's register, parameter passing and run–time stack layout conventions. This means that any data, including structured and dynamic data, can be viewed or set. Other features include breakpoint and assertion modes with which you can control the debugging of the target program. A "transparency" mode allows direct communication with the target system without exiting CrossView Pro. As a Windows debugger it gives you full control of window placement and size. You can customize the interface to suit you requirements.

1.4 Before You Start

This section contains technical information that you may wish to review before installing and starting to use the InterTools software.

1.4.1 Usage Conventions

The common conventions for the InterTools system are described here to avoid duplication in subsequent sections.

In this documentation set, we use M68000 to refer to any microprocessor in the 68000 family. The supported targets within the M68000 microprocessor family are listed at the start of the *C Compiler* chapter in the *InterTools User's Manual*.

1.4.2 InterTools Versions

Every InterTools executable has a version number. To display the version number, invoke the program with the `-0` (zero) option.

The Customer Support department keeps a record of the version numbers of all the executables which have been shipped to you. However, if you report a problem, a support engineer may ask you to run the program with this switch to verify that you are in fact executing the latest version you were sent.

1.4.3 Driver Switches

There are some additional driver switches used primarily for customer support. Please see the *Compiler/Assembler Driver* appendix in the *InterTools User's Manual* for details.

1.4.4 Invocation Conventions

All InterTools programs accept two kinds of arguments: primary arguments and switches.

All switches are preceded by a hyphen ('–'). **All primary arguments must precede all switches and switches are case sensitive.**

For example,

```
form xx.ab -x
```

is correct, but

```
form -x xx.ab
```

is not.

Furthermore, switches cannot be combined as one switch. For example, the following is correct:

```
gsmap object.ln -a -n
```

This is not:

```
gsmap object.ln -an
```

1.4.5 Error Message Output (PC only)

MS–DOS does not provide a mechanism for redirecting error output from the command line. The following two options are accepted by all InterTools programs on the PC:

`-err` [*file*]	Write error messages to file *file*. If *file* does not exist, it will be created. If *file* does exist, it will be overwritten. If *file* is omitted, then error output will be redirected to standard output.
`-err+` [*file*]	Just like `-err`, except output will be appended if *file* exists.

1.5 Additional Help

The InterTools system provides several additional sources for further information on the toolkit, including on–line help and customer support. What follows is a summary of each of these sources with references to more detailed information provided in other sections of the documentation.

1.5.1 Tutorial

The tutorial introduces you to the compiler, assembler and utilities. Sample C and assembly source files are included with this release. By following the tutorial examples while invoking the tools on sample code, you generate listings and learn various switches.

1.5.2 On–line Help

From Command Line

An on–line reference function produces a detailed listing of options available for the compilers and assemblers. For example, if you type:

```
c68000
```

From EDE

All on–line manuals have a corresponding icon located on the right side of the menu bar. Click on the appropriate icon to access.

```
function: compile one or more C programs

usage: c68000 prog.c [prog2.c ...] [options]

Options Summary

Listing Options

-a        expanded source listing, including
#include
             files
-i        interleaved (pseudo-)assembly listing
-l        put all listings in optional list file
name,
             e.g. -l filename
-nf       narrow format pseudo-assembly listing
-p        pseudo-assembly listing
-q        real-assembly listing
-q -i     interlisted real-assembly listing
-s        basic source listing
-x        cross-reference listing

Include Options

-I        specifies user include directories,
             e.g. -I dir1 [dir2 ...]
-S        specifies system include directories,
             e.g. -S dir1 [dir2 ...]

          --Hit <RETURN> to continue; q to quit--
```

Please refer to this manual for help with the software, if the on–line help does not answer your questions.

1.6 Customer Support

This section gives an overview of TASKING customer support. Our goal is to achieve customer satisfaction. The information in this section of the manual explains who is eligible for customer support, outlines the support process, and describes what you can do to expedite this process to help solve your problems as quickly as possible.

1.6.1 Eligibility

The following are eligible for technical support:

- Customers who purchased software directly from TASKING have an active maintenance contract qualify for direct support.
- Customers who purchased their software from Motorola Semiconductor or its authorized distributors within the last year qualify for direct support. Special support arrangements have been made for customers who purchased their software from Motorola Semiconductor. If you purchased your software through Motorola Semiconductor or one of its distributors, please complete the registration form that was included with your release and mail or FAX it to us, so that we can enter you into our customer database.

If you need a copy of the registration form to fill out, you may contact us at: 1–781–320–9400 x239 or, in the continental USA only: 1–800–458–8267.

Note: Customers who purchased software through any other authorized distributor, and have an active maintenance contract with the distributor, will receive direct support through the distributor. When necessary, your distributor will be supported by the TASKING Customer Support organization.

- If you do not have an active maintenance contract and need software support, please contact TASKING, Inc. for information about support contracts and purchasing maintenance, at 1–781–320–9400 x239 or, in the continental USA only, 1–800–458–8267.

1.6.2 If You Have a Problem Using the Software

Please refer to the following pages relating to the country or territory in which you are located:

- USA or Canada: Page 1–24
- Europe: Page 1–25
- Japan: Page 1–26
- All other countries: Page 1–27

Note: There are specific instructions depending on how you obtained the product. Please refer to the instructions within each of the country or territory sections which relate to how you obtained the product. Each section contains specific support instruction for the following methods of purchase:

- Products purchased from a distributor (other than Motorola Semiconductor or one of its distributors).
- Products purchased from TASKING, Inc. or Nihon TASKING directly.
- Products purchased from Motorola Semiconductor or one of its distributors.

Customers located in USA or Canada:

• If the software was purchased from a distributor (other than Motorola Semiconductor or one of its distributors), for support contact:	
Company:	The distributor from which you purchased the product.

• If the software was purchased from TASKING directly, or from Motorola Semiconductor or one of its distributors, for support contact:	
Company:	TASKING, Inc.
Phone:	1–800–458–8267 1–781–320–9400 x239

• If the software was purchased from TASKING directly, or from Motorola Semiconductor or one of its distributors, for support contact:	
Hours:	9:00 A.M. to 5:00 P.M.
Fax:	1–781–320–9743
email:	support@tasking–us.com
Mail:	Customer Support TASKING, Inc. 333 Elm Street Dedham, MA 02026–4530 USA

Customers located in Europe:

• If the software was purchased from a distributor (other than Motorola Semiconductor or one of its distributors), for support contact:	
Company:	The distributor from which you purchased the product.

• If the software was purchased from TASKING directly, for support contact:	
Company:	TASKING BV
Phone:	+31 33 455 8584
Fax:	+31 33 455 1005

• If the software was purchased from TASKING directly, for support contact:	
email:	support_nl@tasking.com
Mail:	TASKING BV P.O. Box 899 3800 AW Amersfoort The Netherlands

Customers located in Japan:

• If the software was purchased from a distributor (other than Motorola Semiconductor or one of its distributors), for support contact:	
Company:	The distributor from which you purchased the product.

• If the software was purchased from TASKING or Nihon TASKING directly, for support contact:	
Company:	Nihon TASKING, K.K.
Phone:	+81 3 3457–6831
Fax:	+81 3 3457–6834
Email:	support_jp@tasking.com
Mail:	Nihon TASKING K.K. Shiba–ST Building, 6th Floor 1–15–13 Shiba Minato–ku, Tokyo 105 Japan

- If the software was purchased from Motorola Semiconductor or one of its distributors, for support contact:

Company:	Nihon TASKING, K.K.
Phone:	+81 3 3457–6831
Fax:	+81 3 3457–6834
Email:	support_jp@tasking.com
Mail:	Nihon TASKING K.K. Shiba–ST Building, 6th Floor 1–15–13 Shiba Minato–ku, Tokyo 105 Japan

Customers located elsewhere:

- Follow the instructions for USA and Canada.

1.6.3 TASKING Customer Support

The functions of the TASKING Customer Support organization are as follows:

- Replace defective media.
- Provide help with the installation and use of your InterTools software. We recommend that you review our documentation and the relevant software and hardware manuals before contacting Customer Support. Customers are assumed to have a basic understanding of their hardware, software, and operating system.
- Work with the customer to provide a suitable solution if the software does not meet the specification described in the InterTools documentation.

- Send updates to customers with active maintenance contracts when new official versions of the software are released.
- Collect suggestions for new features or enhancements for future releases of InterTools software.

To help TASKING Customer Support expedite your calls, please have the following information available when you contact us for help. You may wish to record the information on one of the Software Problem Report (SPR) forms that were included in the Customer Support Packet included in your distribution kit.

- The serial number of your software distribution. This number is printed on the label of the tape, cassette, or first floppy of your software distribution.

  For future convenience, you may wish to record the serial number here:

  Product: _____

  Serial #: _____

- The product name, including host, target processor, and release number.
- The versions of the compiler executables or utilities that you are running.
- The exact command line that you used to invoke our tools when you encountered the problem. Please include all switches.
- The exact error message that was printed. A screen dump will often make this easy to record, and can provide very useful information.

- As concise a test case as possible (with all necessary include files) which displays the problem. This will help us recreate your problem and therefore speed up our response to you. You may email, FAX, or mail us the test case. If your test case is longer than 25 lines, we would appreciate it if you would email the test case or send it on a floppy or tape.
- Any additional information that may be useful in helping to define the problem. Examples include:
  - annotated listings (`.s` listing, for example)
  - your locator command file
  - symbol map file
  - relevant information about your compilation environment
  - the emulator you are using (for CrossView Pro problems)

***REMINDER**: To help us help you, please have the following information ready when you contact us:*

- *Serial #*
- *Product name, host, target, and release #*
- *Compiler version #*
- *Command line*
- *Error message*
- *Test case*
- *Other relevant information*

1.7 Comments, Please

TASKING, Inc. is committed to providing you with the highest quality products and services. To help us in this commitment, we invite your comments and suggestions. We would especially like to know:

- How easy was it to install the software?
- Are you satisfied with the compiler's language features, compilation time, and the speed and size of the generated code?
- Are you satisfied with the assembler's ease of use, and the correctness of the generated object code?
- How easy to use are the programming utilities (Librarian, Linking Locator, Formatter)?
- Are you satisfied with the completeness and efficiency of the run–time library?
- Are you using the product documentation? If so, are you using it as a reference source for specific features or as a complete text?
- How would you rate the level of technical material presented?
- Are there any inaccuracies, or points that need more clarification?
- Are more examples needed? Where, specifically?

Please send your comments and suggestions to:

Product Marketing Manager–InterTools
TASKING, Inc.
333 Elm street
Dedham, Massachusetts 02026–4530
USA

telephone: (781) 320–9400
fax: (781) 320–9212

Thank you for your assistance.

CHAPTER 2

Installation Guide

This chapter contains:

- Introduction
- Installation on Unix Hosts
- Installation on PC Hosts
- Supplementary Reading
- More Help

2.1 Introduction

This document is a guide to the installation of the InterTools system on your PC or Unix host. Please refer to the sections appropriate for your host.

2.2 Installation on Unix Hosts

2.2.1 How to Install the Software

You should log into the system, preferably as root, and create a directory where you will unload the release software. You will need about 1 Megabyte of disk space to unload an assembler release. To unload a compiler release, you will need approximately 4.5 Megabytes of disk space available.

For the examples of installation which follow, we will load the release into a directory named `/usr/local/itools`. If you select a different directory name, you should adjust the instructions accordingly. We will also use the "%" character to indicate the UNIX prompt.

Once you have created the directory for the software, make that your current directory with the UNIX `cd` command. If you are loading the release in `/usr/local/itools`, the following command sequence will create the directory and make it the current directory:

```
% mkdir /usr/local/itools
% cd /usr/local/itools
```

For Solaris and HP 9000/700 hosts, the software is delivered in **tar** format on 4mm, 8mm, or 1/4" data cartridge. To restore the distribution to disk, you must first restore the script which performs the installation. Note that you will need to know the name of the I/O device on which the distribution medium is mounted. You will also need to know whether the I/O device is local (i.e., physically attached to the host computer to which you are logged in) or remote (i.e., attached to some other host computer that is accessible via the UNIX `rsh` facility). If the I/O device is remote, you will need to know the name of the remote host computer.

Note: On HP 9000/700 do not use `rsh` facility in attempt to perform remote shell commands.

For all UNIX hosts, using a local device named `/dev/rmt8`, you would type:

```
% tar xvf /dev/rmt8 install
```

to restore the installation script.

For a remote device name `/dev/rst0` that is attached to host sunkist, you would type:

```
rsh sunkist dd if=/dev/rst0 bs=20b | tar xvBfb - 20 install
```

to restore the installation script.

Now you can run the installation by typing:

```
% ./install
```

You will be prompted for a device name (where the tape is) and the target(s) you wish to install.

To install additional targets or run–time libraries, you may repeat the `install` procedure.

For Solaris, the installation procedure normally includes installation of a license manager, FLEXlm. If your host is Solaris and you already have FLEXlm running on your host, you may wish to install the license manager yourself. You can bypass the installation of FLEXlm during the Toolkit installation by typing:

```
% ./install -nolmgr
```

Refer to the *InterTools FLEXlm Guide* application note of the *InterTools User's Manual* for more information.

Note: You will need the serial # that is printed on the label of your media for customer support. We suggest that you write down your serial number for future use, if necessary. We have provided a space in the Customer Support section of the Introduction to the InterTools User's Manual for this purpose.

For a compiler release on the Solaris, you will also need the license data sheet that is included in your delivery in order to complete the installation.

2.2.2 Components of the Release

The release consists of executables, an installation script, a verification script, the `tutor` directory, and a release notes file. For the **compiler release**, the source for the run–time library is included.

The tutorial files are not necessary for running the InterTools system, and are intended for use with the InterTools Tutorial. After loading the software you will see the following subdirectories and files:

`flexinstall`	License manager installation script (Solaris and HP host only).
`flexlm`	Directory of license manager (Solaris only).
`install`	Installation script.
`readme`	File containing release notes.
`rtlibs`	Compiler run–time library directory (compiler installation **only).**
`tutor`	Directory of files used with the Tutorial. Contains several subdirectories.

`verify`	Verification script.
bin	Directory of executables.

Executables

The x directory contains executables. Most of the executables are names of utilities. The parts of the compiler/assembler and a `driver` program are also included.

Runtime Libraries (Compiler only)

The run–time library contains some low–level support routines that you may need to alter so they work with a particular target system. Source as well as object modules are supplied with the release, so these sources can be modified as necessary. For details on modifying library source files, please read the *Runtime Library* chapter of the *InterTools Reference Manual.*

Tutorial Files

The files in the `tutor` directory are for use with the Tutorial. Users may wish to copy the tutorial files into their own areas. See the *InterTools Tutorial* chapter of the *InterTools Getting Started Manual* for more information on using the tutorial files.

License Manager

The `flexlm` directory contains all necessary support for the license management features. This includes the daemons `lmgrd` and `IMSI`, and a set of end–user license management utilities. Refer to the *InterTools FLEXlm Guide* section in the *Application Notes* chapter of the *InterTools User's Manual* for detailed information regarding license management.

2.2.3 Verifying the Installation

The verification procedure uses a script to verify that the products are working properly. This script is run by the installation program.

```
% verify target_name
```

The script will create a temporary directory named `checkout`, and from there it will assemble, link, locate and format a small assembler test. For **compiler** releases, the script will also compile, link, locate and format a small C language test, producing listing files and hex output. If the verification is successful, the `checkout` directory (and its contents) will be removed.

Verifying Compiler Releases

The following is a sample of successful verification output for a compiler installation and an assembler installation. Both examples were created after performing an installation of the 68000 compiler and assembler.

```
Creating a temporary directory, checkout,
   for running tests
Compiling verify.c with c68000...
Linking verify.ol with the run-time library
   using library index file lib000
Formatting verify.ab in Motorola S records...
Generating Global Symbol Map verify.map ...

Assembling verify.asm with asm68000 ...
Linking verify.ol ...
Formatting verify.ab in Motorola S records ...
Generating Global Symbol Map verify.map ...

Verification of InterTools Toolkit complete.
```

Verifying Assembler Releases

```
Creating a temporary directory, checkout,
   for running tests
Assembling verify.asm with asm68000
Linking verify.ol...
Formatting verify.ab in Motorola S records...
Generating Global Symbol Map verify.map ...

Verification of InterTools Toolkit complete.
```

Conclusion of Verification

If you are satisfied that the products are operating properly, you can now delete the `verify` script:

```
% rm -rf verify
```

Put the directory containing the executables on your search path with the Unix command:

```
% set path = ($path /usr/local/itools/x)
```

For Solaris installations, each user must define an environment variable, `LM_LICENSE_FILE`, to identify the location of the license data file:

```
setenv LM_LICENSE_FILE /usr/local/itools/flexlm/license.dat
```

These lines should be added to the `.login` file of everyone who will use the InterTools software.

2.2.4 Installation Notes

After performing the installation and running the `verify` script, the `x` directory will contain files called, for example, `asm68000` and `c68000` (assuming that the 68000 target was installed). The `install` script creates these links to the `driver` program in the executable directory.

You do not execute the compilers and assemblers directly, rather they are executed by the `driver`, which performs some switch processing and error checking. The same `driver` program operates for all the compilers and assemblers.

The `driver` determines which compiler or assembler it should invoke by checking the name by which driver itself was invoked. **Therefore the `driver` must be linked to specific names, or it will not be able to determine which executable programs to run.** For example, the `driver` divides an invocation name of `asm68000` into `asm` and `68000` from which it knows to run an assembler for the 68000 target. Similarly, `c68000` invokes the C compiler for the 68000 target.

If you are going to move the executables to another directory on the disk, you will have to recreate the appropriate link to the `driver` once you have moved the executables.

2.3 Installation on PC Hosts

2.3.1 How to Install the Software

Note: If you are installing CrossView Pro, the InterTools C Compiler, and EDE, stop here. Go to the Quick Start Guide for installation instructions and for a tutorial on using the tools.

The InterTools software is now installed via a Windows-based installation program. To install InterTools from Windows:

1. Insert the disk labeled **Disk 1** in the floppy drive.
2. Select `File | Run` from the Program Manager menu.
3. Type `a:setup` in the command box, replacing `a:` with the name of your floppy drive, if necessary.
4. Click on `OK`.
5. Then follow the instructions in the setup program.

The new installation program does not install any run-time libraries by default. The type and number of run-time libraries installed is fully configurable, and is selected via the InterTools Base Components Options dialog. Compiler and assembler driver programs (`c68xxx.exe` and `asm68xxx.exe`) will be installed based on the run-time library selections; therefore, you must select at least one run-time library in order to be able to run the InterTools compiler.

For an assembler release, you will need about two megabytes of disk space to unload the entire package onto a hard disk with a cluster size of 16 KB. The amount of space required will vary depending on the cluster size of your hard disk.

For a compiler release, you will need about 12 megabytes of disk space on your hard disk, assuming that one library is installed; the exact amount depends upon how many run–time libraries you choose to install and the cluster size of your hard disk.

Note: You will need the serial # that is printed on the label of the first diskette for customer support. We suggest that you write down your serial number for future use.

2.3.2 Release Notes

After the installation has been completed, there will be a file in the top–level directory called `readme.txt`. This file contains information on new or updated features and quality improvements, and reports known problems and bugs fixed.

2.4 Supplementary Reading

You should read the *InterTools User's Manual*, paying particular attention to the *Linking Locator* chapter. The *InterTools Tutorial* in this manual provides information for new InterTools users.

2.5 More Help

If you have any difficulty during the installation of your product, please contact the TASKING Customer Support staff. We can be reached at (800) 458–8276, 9:00 AM to 5:30 PM, Eastern Time.

CHAPTER 3

InterTools Tutorial

This is the InterTools tutorial, a step–by–step series of examples and exercises designed to teach you how to use your Toolkit. The examples vary in difficulty from simple to advanced, allowing you to progress to more advanced functions. The last section of this chapter gives an introduction to system building concepts so that new users can get started quickly.

3.1 Introduction

Our goal in this tutorial is to teach you how to use the InterTools package to build an executable program from your C source and/or assembly language programs. The programs which comprise the InterTools package are described in the *The InterTools System* section in the *Introduction* chapter of this manual. We approach the tutorial by guiding you through examples with the tools, using sample programs included with the InterTools software. In the later examples we introduce a few advanced topics that will help you understand this manual. Throughout the tutorial, we follow document conventions described in the front of the manual.

In this tutorial, we will be compiling and assembling files for the M68000 family. You will also be introduced to TASKING's graphical user interface, EDE, which gives you point–and–click control over the whole development process. We realize that you may have only one target available to you. The example output is included in such a way as to be useful for a variety of M68000 family targets. We encourage you to run as many examples as are applicable, substituting your own target in place of the target listed in each example. Instructions for invoking the compiler and assembler for each target are described below. You can find additional details throughout this manual.

The first part of the tutorial emphasizes the use of the InterTools components. The second part of this chapter introduces important topics such as system initialization, memory management, and linking C and assembly language.

3.2 Finding the Programs and Setting Up the Path

The InterTools release contains many different files. Some are executables: files that make up the compiler, assembler, and utilities. Others contain sample programs, which we use in this tutorial to teach the InterTools system. The files are organized into directories and subdirectories for easy reference. Before you can begin to use InterTools, you must know where the directories are located and what is contained in each.

Note: This getting started guide will include some information about C++. For more information, consult your *C++ Compiler User's Guide*.

For discussion in this manual, we assume that the InterTools release was loaded on drive `c:` in the directory `itools` for PC hosts or in

`/usr/local/itools` for Unix hosts. If so, the InterTools executables will probably be in `c:\itools\bin` or `/usr/local/itools/bin`, respectively. You may want to modify your search path to find the InterTools programs. On the PC, one way to do so is to edit your `autoexec.bat` file to include `c:\itools\bin` in the path portion of the file, so it reads as follows:

```
path c:\bin; c:\itools\bin; ...
```

On a Unix host, you can modify your search path by editing your `.cshrc` file to include `/usr/local/itools/bin` in the `set path=` portion of the file, so it reads as follows:

```
set path = (. /usr/local/itools/bin ...)
```

Your system administrator may have loaded the InterTools release in another directory. If a different directory is being used, substitute its name.

You will find at least three subdirectories under the directory where the InterTools release was loaded. To verify this on the PC, type:

```
cd \
cd itools
dir
```

or, on a Unix host, type:

```
cd /usr/local/itools
ls
```

On the PC, you will see:

```
RTLIBS <DIR>
TUTOR  <DIR>
bin      <DIR>
```

or, on Unix you will see:

```
rtlibs tutor bin
```

if your system administrator has not renamed any of the subdirectories. For the purpose of this tutorial, we will assume that none of the directories has been renamed.

Any additional directories or executables are for installation only, and will not be discussed in this tutorial. The InterTools directories and their contents are described in the next section.

3.2.1 bin Directory

The directory `bin` contains the executables, or programs which run the compiler, assembler and utilities. Some of these executables are user–invoked: you must specify that the file be executed. Others are called automatically. You never enter these automatically invoked files in any commands. The exact contents of the `bin` directory will depend on which targets will be used for compiling and assembling. These targets were chosen at installation time. Executables to compile or assemble code for additional targets may be loaded as needed.

To view the contents of `bin` on the PC, type:

```
dir bin
```

or, on a Unix host, type:

```
ls bin
```

A brief description of each file in this directory can be found below:

3.2.2 Assembler Files

`AS68K.EXE` **(PC only)** `as68k` **(Unix only)**	Assembler executable. This file is invoked automatically when you invoke one of the target–specific programs below. You do not invoke this executable file directly.

There will also be at least one of the following files, depending on the target(s) chosen at installation time:

Target Selected	PC File Created	Unix File Created
MC68000	ASM68000.EXE	asm68000
MC68EC000	ASM68000.EXE	asm68000
MC68008	ASM68000.EXE	asm68000
MC68010	ASM68010.EXE	asm68010
MC68020	ASM68020.EXE	asm68020
MC68EC020	ASM68020.EXE	asm68020
MC68030	ASM68030.EXE	asm68030
MC68EC030	ASMEC030.EXE	asmec030
MC68040	ASM68040.EXE	asm68040
MC68EC040	ASMEC040.EXE	asmec040
MC68302	ASM68302.EXE	asm68302
MC68330	ASM68332.EXE	asm68332
MC68331	ASM68332.EXE	asm68332
MC68332	ASM68332.EXE	asm68332
MC68340	ASM68340.EXE	asm68340
MC68360	ASM68360.EXE	asm68360
MC68060	ASM68060.EXE	asm68060

3.2.3 Compiler Files

There will be at least one of the following files, depending on the target(s) chosen at installation time:

Target Selected	PC File Created	Unix File Created
MC68000	C68000.EXE	c68000
MC68EC000	C68000.EXE	c68000
MC68008	C68000.EXE	c68000
MC68010	C68010.EXE	c68010
MC68020	C68020.EXE	c68020
MC68EC020	C68020.EXE	c68020
MC68030	C68030.EXE	c68030
MC68EC030	CEC030.EXE	cec030
MC68040	C68040.EXE	c68040
MC68EC040	CEC040.EXE	cec040
MC68302	C68302.EXE	c68302
MC68330	C68332.EXE	c68332
MC68331	C68332.EXE	c68332
MC68332	C68332.EXE	c68332
MC68340	C68340.EXE	c68340
MC68360	C68360.EXE	c68360
MC68060	c68060.EXE	c68060

The following executables are invoked automatically by the compiler executables above:

`BC68000.EXE` **(PC only)** `bc68000` **(Unix only)**	Compiler back end.
`CPF.EXE` **(PC only)** `cpf` **(Unix only)**	C compiler preprocessor and front end.
`DRIVER.EXE` **(PC only)** `driver` **(Unix only)**	Invokes the proper target–specific compiler and assembler files.
`FLOW.EXE` **(PC only)** `flow` **(Unix only)**	Global optimizer.
`INTERL.EXE` **(PC only)** `interl` **(Unix only)**	Used to produce interleaved assembly listings, debugging symbols, and cross–reference listings
`MERGE.EXE` **(PC only)** `merge` **(Unix only)**	respectively. These executables are run automatically when the proper compiler
`XREF.EXE` **(PC only)** `xref` **(Unix only)**	switches are specified.

3.2.4 C++ Compiler Files

There will be at least one of the following files, depending on the target(s) chosen at installation time:

Target Selected	PC File Created	Unix File Created
MC68000	CP68000.EXE	cp68000
MC68EC000	CP68000.EXE	cp68000
MC68008	CP68000.EXE	cp68000
MC68010	CP68010.EXE	cp68010
MC68020	CP68020.EXE	cp68020
MC68EC020	CP68020.EXE	cp68020
MC68030	CP68030.EXE	cp68030
MC68EC030	CPEC030.EXE	cpec030
MC68040	CP68040.EXE	cp68040
MC68EC040	CPEC040.EXE	cpec040
MC68302	CP68302.EXE	cp68302
MC68330	CP68332.EXE	cp68332
MC68331	CP68332.EXE	cp68332
MC68332	CP68332.EXE	cp68332
MC68340	CP68340.EXE	cp68340
MC68360	CP68360.EXE	cp68360
MC68060	cP68060.EXE	cp68060

3.2.5 Utilities

`FORM.EXE` **(PC only)** `form` **(Unix only)**	Object module formatter.
`GSMAP.EXE` **(PC only)** `gsmap` **(Unix only)**	Utilities which generate various listings of symbol information.
`SYMLIST.EXE` **(PC only)** `symlist` **(Unix only)**	
`LIBR.EXE` **(PC only)** `libr` **(Unix only)**	Librarian utility.
`LLINK.EXE` **(PC only)** `llink` **(Unix only)**	Linking locator.
`OLSIZE.EXE` **(PC only)** `olsize` **(Unix only)**	Utility which lists the total size of a set of object modules.

3.2.6 C++ Utilities

`LDRIVER.EXE`	C++ Linker
`EDG_MUNCH.EXE`	Linker Utility
`EDG_PRELINK.EXE`	Linker Utility
`EDG_DECODE.EXE`	C++ Name Demangling

3.2.7 rtlibs Directory

The compiler runtime libraries are contained in the directory `rtlibs`. The runtime libraries are necessary for linking, and contain source files, include files, and compiled or assembled runtime library routines, which resolve external references.

To see what files are in the `rtlibs` directory on the PC, type:

```
dir rtlibs
```

or, on a Unix host, type:

```
ls rtlibs
```

You will see at least one of the following subdirectories, depending on the target(s) chosen during the installation:

Target Processor	Floating Point Package	Target Board	Library Directory
MC68000	SW	—	`lib000`
MC68EC000	SW	—	`lib000`
MC68008	SW	—	`lib000`
MC68010	SW	—	`lib010`
MC68020	SW	—	`lib020s`
MC68020	HW	—	`lib020h`

Target Processor	Floating Point Package	Target Board	Library Directory
MC68EC020	SW	—	`lib020s`
MC68EC020	HW	—	`lib020h`
MC68030	SW	—	`lib030s`
MC68030	HW	—	`lib030h`
MC68EC030	SW	—	`libe30s`
MC68EC030	HW	—	`libe30h`
MC68040	on–chip f.p.	—	`lib040`
MC68EC040	SW	—	`libe40`
MC68302	SW	—	`lib302`
MC68302	SW	M68302ADS Development System using parallel I/O	`lib302ap`
MC68302	SW	M68302ADS Development System using trap I/O	`lib302at`
MC68330	SW	—	`lib332`
MC68331	SW	—	`lib332`
MC68332	SW	—	`lib332`
MC68340	SW	—	`lib340`
MC68340	SW	M68340BCC Development System	`lib340b`

Target Processor	Floating Point Package	Target Board	Library Directory
MC68360	SW	—	`lib360`
MC68360	SW	M68360QUADS Development System	`lib360b`
MC68060	HW	—	`lib060`

Below each of these directories on the PC are the following subdirectories (if your system administrator has not renamed them):

```
INC    <DIR>
LIB    <DIR>
SRC    <DIR>
```

The `INC` directory contains the library include files. The `LIB` directory contains the library index files and object files. The `SRC` directory contains the library source files.

The C++ compiler contains the additional directories `cppinc` and `cpplib`.

3.2.8 examples Directory

The `examples` directory contains two subdirectories, `tutor` and `cpp`. The `tutor` subdirectory contains files of sample C programs and assembly language routines which you use throughout the tutorial. The cpp subdirectory contains sample C++ programs.

The list below summarizes the source files contained in the tutor subdirectory:

`adexp.68k`	We will use this assembly language program to demonstrate use of the assembler.
`circle.c`	This program contains a function to compute thearea of a circle given its radius. this file demonstrates floating point features.
`fact.c`	This program includes a function to compute the factorial of a number. It has external references to the routines in `circle.c` and `adexp.68k`. Also, it contains external references to be resolved by linking to the runtime library.
`fpneg.68k`	This assembly language routine resolves an external reference in `fact.c`. We use the program to demonstrate the linking locator's function of resolving references.
`inc`	The `inc` directory contains the include files `dargstac.h`, `def.h`, and `ret_doub.h` as included source. In the examples, you will specify this directory in order to find files named in `#include` directives.
`link.lst`	We will use this file to demonstrate a linking feature of the linking locator.

`loc.cmd`	This is a locator command file which demonstrates a locating feature of the linking locator utility.

`asmfile.68k`
`cfile.c`
`main.c`
`loc-main.cmd`
	These files are used in the *Introduction to System Building Concepts* section

Before you proceed with the tutorial, we recommend that you make your own copy of the `tutor` subdirectory, its contents, and its subdirectories. InterTools programs create files as you use the tools, and you may want to keep the tutorial intact for future use.

For the remainder of the tutorial, we assume that you have set your working directory to your copy of the `tutor` sub0directory.

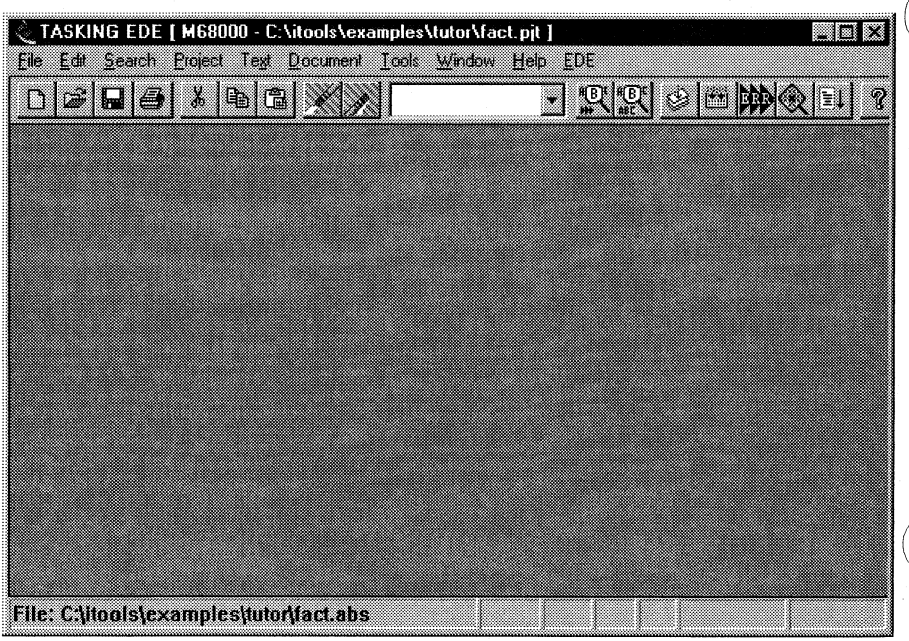
3.3 Invoking the Tools

3.3.1 Invoking the Tools from EDE

EDE's point and click intertace allows you to compile, and build your project without using the command line.

Select EDE from the InterTools menu in the Program Manager to open EDE.

This is what you see:

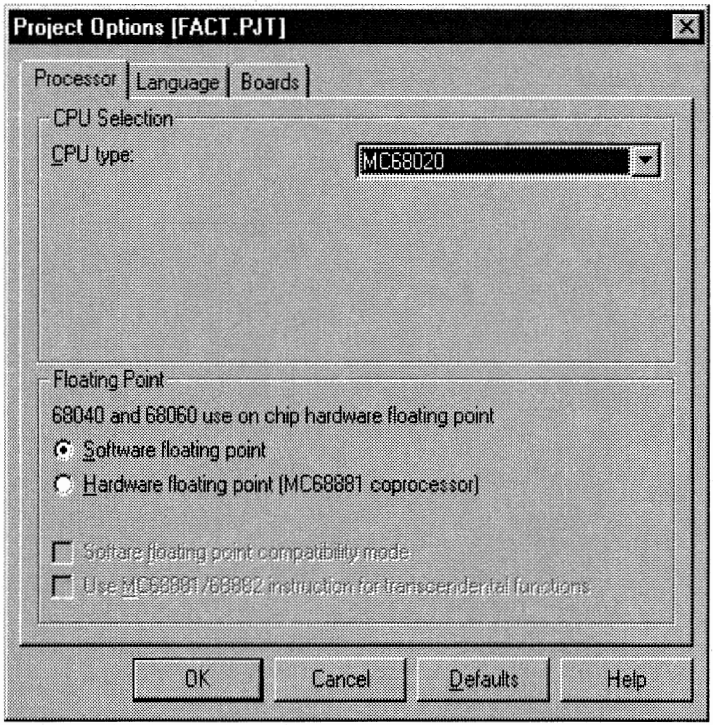

TASKING EDE [M68000 - C:\itools\examples\tutor\fact.pjt]
File Edit Search Project Text Document Tools Window Help EDE
File: C:\itools\examples\tutor\fact.abs

Follow this procedure to set the options for the *fact.pjt* project.

1. Select **Project | \itools\examples\tutor\fact.pjt**
   The files that make up *fact.pjt* appear in EDE
2. Select **EDE | Build Options** and check the "show command line options at the bottom of the tool tab" box if it is not already checked. Click OK

3. Select **EDE | Project Options**

   The Project Options dialog box appears.

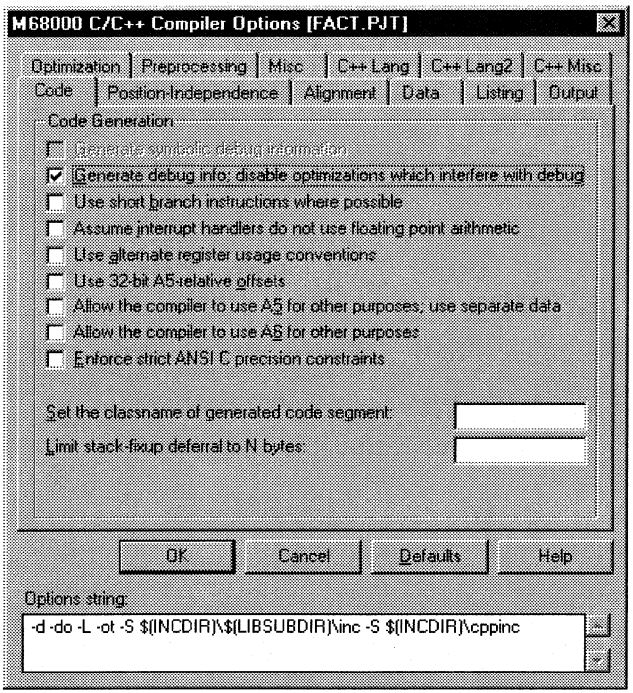


4. Select the **Processor** tab and select processor 68020.
5. Select the "Software floating point" radio button in the Floating Point section.
6. Select the **Language** tab and select the "C only (no C++ modules)" radio button.
7. Select the "Long integer data type" radio button. Click OK.

   Once the project options are set, you can set the remaining options. To set the other options:

1. Select **EDE | C/C++ Compiler Options | Project Options.**

   The M68000 C/C++ Compiler Options dialog box appears.



2. Select the compiler options you want.

   NOTE: A complete list of all the EDE compiler options appears in the EDE Read Me First file that comes with this product.

Notice that the command options, as you set or remove them, appear in the box at the bottom of the dialog.

3. Select the options you want from the **Assembler Options** and **Linker/Formatter Options** using the same method as with the compiler options.

4. Select the options you want from CrossView Pro Options by selecting **EDE | CrossView Pro Options.**

Follow this procedure to compile and build a program:

1. Click on the Build icon.

2. Click the **Flyswatter** icon to bring up CrossView Pro.

3.3.2 Invoking the Tools Using Command Line

All of the InterTools programs follow a general three–part syntax. First, name the program you want to run. Next, name the primary arguments. Finally, append options or switches.

The program named first on the command line is the InterTools command that acts upon the input. For instance, the command `c68000` compiles the C source code in the input file(s), creating one or more object modules for the MC68000 target. The command `c68020` compiles source code for the MC68020 target. All the examples shown in this tutorial use the MC68020 target. You should substitute your own target on the command line (for examples that are applicable to your target) when trying the sample commands described below.

The primary arguments are usually names of input files. The files may be source, object modules, or absolute files, depending on the executable. We will discuss each of these files in the examples.

To generate listings or specify particular operations of the program, append the necessary switches to the invocation line. Switches always begin with a hyphen, "–". While you can append multiple switches,

you must include the hyphen for each. For example, the following syntax is correct:

```
c68020 fact.c -s -l
```

The combination below is incorrect:

```
c68020 fact.c -sl
```

Note that due to the page width limitations of this manual, we may present a single invocation line as two or more lines. You should, however, type the invocation all on one line.

3.4 Tutorial Examples

The remainder of this tutorial is dedicated to running the InterTools programs on the sample source programs provided with the release. The examples cover the following topics:

Example 1: Building Your First Application Executable

Example 2: Listings and Non–Default Output Files

Example 3: Non–Default Memory Models and Linking Options

Example 4: Locator Options

Example 5: Formatting Options and Saving Symbol Information

Note: The remaining examples all use PC–style directory pathnames; that is, they use backslashes (\) and start with `itools`. If your host is Unix, you should use forward slashes (/) and directory names starting with something like `/usr/local/itools` instead.

3.4.1 Example 1: Building Your First Application Executable

In this example you will do the following:

- Compile a C source program.
- Link, ROM process and locate the compiled object module.
- Format the absolute module in the default format for the 68000 family.

STEP 1

Compile a C source program, `circle.c`. In our example, we compile for the MC68020 target. Type:

```
c68020 circle.c
```

STEP 2

Link, ROM process and locate the object module `circle.ol`. Search the runtime library specified by the library index file `itools\rtlibs\lib020s\lib\lib020s` to resolve references. Write output to a file with the default suffix. Type:

```
llink circle.ol -L itools\rtlibs\lib020s\lib\lib020s -o
```

STEP 3

Format the absolute module `circle.ab` using the default format:

```
form circle.ab
```

Explanation

In STEP 1 we compiled `circle.c` and created an object module for the MC68020 target. The InterTools command `c68020` compiles for the MC68020 target. (You can substitute a different compiler command name for your own target if you are trying this example.) You named `circle.c` as the input file. By default the compiler writes the object module to output to a file with the suffix `.o1`. In this example, the compiler wrote the object module `circle.c`. To verify this, list the directory. You should see `circle.c` among the files.

In STEP 2 we linked, ROM processed and located the object module `circle.c`. The InterTools executable `llink` invokes the linking locator.

The `-L` switch specifies the library or libraries to be searched to resolve references during linking. In this case, we specified the library `itools\rtlibs\lib020s\lib\lib020s`, for the software floating point 68020 library. This is the appropriate library for the options we chose, since by default the compiler generates instructions for software floating point operations. Some targets do not have separate software and hardware floating point libraries. The organization of the compiler runtime library is described fully in the *Linking Locator* chapter in the *InterTools User's Manual.* If you are trying this example, substitute the name of the appropriate runtime library for your target.

When locating this object module, the linking locator assigns MC68020 memory locations to code and data in the object module. By default, it assigns code and data segments to consecutive addresses, beginning at zero.

The `-o` switch directs the linking locator to write output to a file, rather than the terminal. The output is called absolute because the addresses are fixed, or completely located, and is therefore named with the `.ab` suffix. You should see the file `circle.ab` in your directory listing. If we had not appended the `-o` switch, the linking locator would have directed the output to the screen rather than saving it in a file.

In STEP 3 we formatted `circle.ab` using the default 68000 family format, Motorola S–records. InterTools formats the named absolute module `circle.ab`, and by default, names the output file by appending the default `.hex` suffix. You should see the file `circle.ab` in your directory list.

This Motorola S–record hex file is the final result of the InterTools processing chain. It can be loaded into your target system in a variety of ways and then executed. Refer to the *Downloading* application note in the *InterTools User's Manual* for more details.

3.4.2 Example 2: Listings and Non–Default Output Files

In this example you will do the following:

- Generate a source listing.
- Generate and write the following listings to the file `fact.mny`: pseudo–assembly interleaved with source code in narrow format, cross–referenced.
- Assemble two assembly language source files and generate a listing for one file which shows both primary and included source, and expanded macros.
- Link, ROM process and locate the compiled and assembled object modules.
- Format the resulting absolute `.ab` file, and direct the output to a non–default file.

STEP 1

Compile C source code in `circle.c` for the MC68020 target and generate a source listing. Type:

```
c68020 circle.c -s
```

STEP 2

Compile C source code in `fact.c` for the MC68020 target and generate a narrow format pseudo–assembly listing interleaved with source code, and a cross–reference listing. Write both listings to `fact.mny`. Type:

```
c68020 fact.c -nf -x -l fact.mny
```

STEP 3

Assemble the assembly language routines `adexp.68k` and `fpneg.68k` for the MC68020 target. Search the `inc` directory for include files. Generate a listing for `adexp.68k` which shows included source, and expanded macros. Type:

```
asm68020 adexp.68k -a -m -I inc
asm68020 fpneg.68k
```

STEP 4

Link, ROM process and locate the object modules `fact.o1`, `circle.o1`, `fpneg.o1` and `adexp.o1`. Search the library

`itools\rtlibs\lib020s\lib\lib020s` to resolve references. Write output to the default file. Type:

```
llink fact.ol circle.ol adexp.ol fpneg.ol -L
   itools\rtlibs\lib020s\lib\lib020s -rs idata -o
```

Note that the entire `llink` command above should be typed on one line.

STEP 5

Format the absolute file `fact.ab` in the default format for the Motorola family, packed Motorola. Write output to a non–default file. Type:

```
form fact.ab -o output.hex
```

Explanation

In STEP 1 we compiled the C source program `circle.c` for the MC68020 target. By appending the `-S` switch to the invocation line,

we generated a source listing with the default suffix `.lis`. The listing `circle.lis` appears below:

```
circle.c                        Apr 30 1991  14:09:13
                                             PAGE    1

  1     /*
  2     @(#)m/r/tutor/circle.c  1.5
  3     */
  4     /* calculates the area of a circle */
  5
  6     float pi = 3.1416;
  7
  8     float circle(radius)
  9        double radius;
 10     {
 11        float answer;
 12
 13        /* pow is a function supplied *
 14        /* in the C runtime library. */
 15        answer = pi * pow(radius,2.0);
 16        return(answer);
 17     }
```

In STEP 2 we compile the C program `fact.c` for the MC68020 target. Because the InterTools compiler converts source code directly to object code, it normally bypasses generating an assembly language representation. To see the C source program as it would appear in assembly language, InterTools allows you to generate a pseudo–assembly listing. This pseudo–assembly listing can also show the C source interleaved with the generated code. To make an interleaved listing which uses 80 column lines rather than the default of 132, we appended the `-nf` switch. A portion of the interleaved listing from `fact.mny` appears below:

```
*35          sum = 0;
        CLR.L   -20(A6)                 42aeffec        .loc 000000 + P1

*36
*37          for (loopvar = 1; loopvar < 8; ++loopvar) {
        MOVEQ.L #1,D2                   7401            .loc 000004 + P1

*(code hoisted from following statement)
        LEA.L   -14(A6),A1              43eefff2        .loc 000006 + P1
L20001
*38              table[loopvar] = factorial(loopvar);
        MOVE    D2,-(A7)                3f02            .loc 00000a + P1
        JSR     _factorial              4eb9________    .loc 00000c + P1
        ADDQ.L  #2,A7                   548f            .loc 000012 + P1
        MOVE    D0,(A1)                 3280            .loc 000014 + P1

*39              sum += table[loopvar];
        MOVE    (A1)+,-(A7)             3f19            .loc 000016 + P1
        JSR     __Itof                  4eb9________    .loc 000018 + P1
        ADDQ.L  #2,A7                   548f            .loc 00001e + P1
        MOVE.L  D0,-(A7)                2f00            .loc 000020 + P1
        MOVE.L  -20(A6),-(A7)           2f2effec        .loc 000022 + P1
        JSR     __Fadd                  4eb9________    .loc 000026 + P1
        ADDQ.L  #8,A7                   508f            .loc 00002c + P1
        MOVE.L  D0,-20(A6)              2d40ffec        .loc 00002e + P1

*(see line 37)
        ADDQ    #1,D2                   5242            .loc 000032 + P1
        CMPI    #8,D2                   0c420008        .loc 000034 + P1
        BLT.S   L20001                  6dd0            .loc 000038 + P1

*40          }
```

A cross–reference listing is a table which shows all user–defined types, variables and constants, the line numbers in the source code where they are defined, and any line numbers where they are referenced. The -x switch appended to the invocation line generated

this listing. Part of the cross–reference listing portion of `fact.mny` appears below:

```
Apr 24 1997  10:29:49   CROSS-REFERENCE  :  fact.c
  PAGE    1

ENUMTYPE
    Def :  fact.c              8
    Ref :  fact.c             14

FPNEG
    Def :  * undefined *
    Ref :  fact.c             54

RECTYPE
    Def :  fact.c             15
    Ref :  fact.c             18      19

a
    Def :  fact.c             11
    Ref :  fact.c             47

b
    Def :  fact.c             12

blue
    Def :  fact.c              8
    Ref :  fact.c             18      46
```

To write output to a non–default file, we appended the `-l` switch with the desired file name, `fact.mny` as an argument. The compiler wrote each listing to `fact.mny` consecutively.

In STEP 3 we assembled the assembly language routines `adexp.68k` and `fpneg.68k` for the MC68020 target. The `asm68020` InterTools executable assembled, for the MC68020 target, the input files `adexp.68k` and `fpneg.68k`. You can substitute a different assembler executable name to assemble for your target.

The source program `adexp.68k` names another file to be included upon assembly. By default, the assembler searches the working directory for include files. In this tutorial, the named include file resides in the `inc` subdirectory. To search outside the working directory, we appended the `-I` switch with the directory inc as an argument.

The `-a` switch generates an assembler listing that contains primary and included source. By default, the listing file has the `.lis` suffix. A portion of the listing `adexp.lis` appears below. Note that lines #13 and #14 name `dargstac.h` and `ret_doub.h` as included source. The following portion of `adexp.lis` shows an assembler listing that contains the expansion of the first include file source.

```
9       0           |    *     EXTERNAL PROCEDURES: DPEXP
10      0           |    *     EXTERNAL DATA:
11      0           |
*****************************************************************
12      0           |    *
13      0           |              include 'dargstac.h'
1     1 0           |
*****************************************************************
2     1 0           |    *     include file dargstac.h for tutor directory
3     1 0           |    *
4     1 0           |    *     @(#)m/r/tutor/tutor682/inc/dargstac.h
1.1
5     1 0           |
*****************************************************************
6     1 0           |    *
7     1 0    [$4]   |    regw     set      4       register width
8     1 0    [$E]   |    regnum   set      14      number of registers saved
9     1 0    [$38]  |    regspc   set      regw*regnum  amount of space
                    |                                 taken by saved regs
10    1 0    [$0]   |    numbyt   set      0       init numbyt
```

The `-m` switch causes the assembler to include macro expansions in the listing file. Macros provide a shorthand means to invoke a series of assembly language source statements that appear a number of times throughout a program. Rather than writing the consecutive lines of code at every appropriate point in the program, you can name the consecutive lines as a "macro," then invoke the macro with a single line of code. Note that line #12 of the `dargstac.h` included source file defines a macro. The macro ends at line #23. The macro is invoked in the file `adexp.68k`, which includes `dargstac.h`. For more information on macros, please refer to the *InterTools Reference Manual*. Macro expansions show the macro contents read by the assembler.

In STEP 4 we linked, ROM processed and located the compiled and assembled `.o1` files `fact.o1`, `circle.o1` and `adexp.o1`. We named each input object module, separating file names with a space.

The `-rs idata` switch causes the linking locator to perfor ROM processing. ROM processing is described fully in the *Linking Locator* chapter in the *InterTools User's Manual*. Briefly, ROM processing provides a way for the program to initialize its global data at run time. Initialization is done by copying from a ROM–resident initialization segment (created by the ROM processor) into the segment. By default the InterTools compiler allocates all initialized data in the idata segment. In this example, by supplying the `idata` argument to the `-rs` switch, we create a new segment with the initialization values from the idata segment.

To resolve external references, we first searched the library file `itools\rtlibs\lib020s\lib\lib020s` and then we located by default.

The linking locator wrote output to the default file `fact.ab`, using the first file name listed, `fact.o1`, to form the root, and appended the default `.ab` suffix.

In STEP 5, we entered the following invocation:

```
form fact.ab -o output.hex
```

This invocation formatted the absolute file `fact.ab` in the default format, packed Motorola, and wrote the output to a non–default file, `output.hex`, by appending the `-o` switch with the argument `output.hex`.

3.4.3 Example 3: Non–Default Memory Models and Linking Options

This example illustrates various switches which affect the compiler's choice of generated code. Due to the many approaches, we have

divided the example into two sections: floating point compilations, and compilations with the long integer datatype option. We will do the following:

- Compare compilations for the MC68020 target using hardware and software floating point.
- Link and ROM process multiple files named in a single file.
- Compile using the long integer datatype option.
- Link and locate using a long integer library.
- Format the absolute files.

Floating Point Compilations

STEP 1

Compile and generate an interleaved assembly and source listing for the MC68020 target using software floating point. Write the listing to `circle.sw`. Type:

```
c68020 circle.c -nf -l circle.sw
```

STEP 2

Compile and generate an interleaved assembly and source listing for the 68020 target using hardware floating point. Type:

```
c68020 circle.c -nf -h
```

STEP 3

Compile `fact.c` for the MC68020 target using hardware floating point and assemble `fpneg.68k`. Type:

```
c68020 fact.c -h
asm68020 fpneg.68k
```

STEP 4

Link only the three `.ol` files `fact.ol`, `circle.ol` and `fpneg.ol` named in a single file, `link.lst`.

Link with the MC68020 hardware floating point runtime library. Type:

```
llink -i link.lst -rs idata
    -L itools\rtlibs\lib020h\lib\lib020h -lo -o
```

Explanation

In STEP 1, we compiled `circle.c` for the MC68020 target using software floating point. The assembly and source code listing from file `circle.sw` appears below. Note the code size shown at the bottom:

```
*13             /* pow is a function supplied */
*14             /* in the C runtime library. */
*15         answer = pi * pow(radius,2.0);
        MOVE.L  _pi-data(A5),D1         222d____        .loc 000000 + P1
        MOVE.L  D1,-(A7)                2f01            .loc 000004 + P1
        CLR.L   -(A7)                   42a7            .loc 000006 + P1
        MOVE.L  #:40000000,-(A7)        2f3c40000000    .loc 000008 + P1
        MOVE.L  12(A6),-(A7)            2f2e000c        .loc 00000e + P1
        MOVE.L  8(A6),-(A7)             2f2e0008        .loc 000012 + P1
        JSR     _pow                    4eb9________    .loc 000016 + P1
        ADDA    #16,A7                  defc0010        .loc 00001c + P1
        MOVE    D0,-(A7)                3f00            .loc 000020 + P1
        JSR     __Itof                  4eb9________    .loc 000022 + P1
        ADDQ.L  #2,A7                   548f            .loc 000028 + P1
        MOVE.L  (A7)+,D1                221f            .loc 00002a + P1
        MOVE.L  D0,-(A7)                2f00            .loc 00002c + P1
        MOVE.L  D1,-(A7)                2f01            .loc 00002e + P1
        JSR     __Fmul                  4eb9________    .loc 000030 + P1
        ADDQ.L  #8,A7                   508f            .loc 000036 + P1
        MOVE.L  D0,-4(A6)               2d40fffc        .loc 000038 + P1

*16         return(answer);
        UNLK    A6                      4e5e            .loc 000040
        RTS                             4e75            .loc 000042

* Function size = 68

* bytes of code = 68
* bytes of idata = 4
* bytes of udata = 0
* bytes of sdata = 0
        XREF    _pow
        XREF    __Itof
        XREF    __Fmul
        XREF    data
```

By default, the InterTools 68000 family compiler generates software floating point instructions.

In STEP 2 we compiled using hardware floating point and generated an interleaved listing in narrow format. The `-h` switch directs the compiler to use MC68881 instructions to perform floating point operations. If your target is not equipped with the MC68881, do not use the `-h` switch.

By default, the compiler names the interleaved listing by appending the `.psa` suffix. A portion of `circle.psa` appears below:

```
*13             /* pow is a function supplied */
*14             /* in the C runtime library. */
*15         answer = pi * pow(radius,2.0);
        FMOVE.S _pi-data(A5),FP4        f22d4600____    .loc 000000 + P1
        FMOVE.X FP4,-(A7)               f2276a00        .loc 000006 + P1
        CLR.L   -(A7)                   42a7            .loc 00000a + P1
        MOVE.L  #:40000000,-(A7)        2f3c40000000    .loc 00000c + P1
        MOVE.L  (__R1+32)(A7),-(A7)     2f2f____        .loc 000012 + P1
        MOVE.L  (__R1+32)(A7),-(A7)     2f2f____        .loc 000016 + P1
        JSR     _pow                    4eb9________    .loc 00001a + P1
        ADDA    #16,A7                  defc0010        .loc 000020 + P1
        FMOVE   D0,FP4                  f2005200        .loc 000024 + P1
        FMOVE.X (A7)+,FP0               f21f4800        .loc 000028 + P1
        FSGLMUL.X FP0,FP4               f2000227        .loc 00002c + P1
        FMOVE.X FP4,FP1                 f2001080        .loc 000030 + P1

*16         return(answer);
        FMOVE.X FP1,FP0                 f2000400        .loc 000034 + P1
        FMOVE.X (A7)+,FP1               f21f4880        .loc 00003c
        RTS                             4e75            .loc 000040

* Function size = 66

* bytes of code = 66
* bytes of idata = 4
* bytes of udata = 0
* bytes of sdata = 0
        XREF    _pow
        XREF    data
        END
*17     }
```

Note the figure of bytes of code shown near the end of the listing for comparison with the software floating point example.

In STEP 3, we compiled `fact.c` using hardware floating point. To do so, we used the hardware floating point switch, `-h`.

In this step, we also assembled `fpneg.68k`. Due to the nature of assembly language, there is no need to append a switch to specify floating point type, so we assembled `fpneg.68k` normally for the MC68020 target, by using the `asm68020` command.

In STEP 4, we linked the three object modules, but rather than listing each file name separately, we listed the names of the object language files in a single file, `link.lst`, included with this release. By appending the `-i` switch and file name, we directed the linking locator to read the files named in `link.lst`. If you list the contents of `link.lst`, you will see the three file names.

Because the compiled object modules use hardware floating point, we linked with the hardware floating point library, `itools\rtlibs\lib020h\lib\lib020h`.

The `-rs` switch with the `idata` argument ROM processes the idata segment. To link only, we appended the `-lo` switch, This switch bypasses the locate step, leaving a relocatable object module. Linked and ROM processed output is named, by default, with the `.rmp` suffix. By using the root of the first object module listed, and appending the `.rmp` suffix, the linking locator wrote the linked output to `fact.rmp`.

The Long Integer Datatype Option

Compile for the MC68020 target using the long integer datatype option. Link, locate and format. Type:

```
c68020 circle.c -L
llink circle.o1 -L
    itools\rtlibs\lib020s\lib\lib020s.l -o
```

Explanation

In the previous step, we compiled using the long integer option datatype option. The long integer switch, `-L`, causes the compiler to assign integers four bytes of memory, and short two bytes. The default is to assign integer two bytes of memory and shorts one byte. When invoking the linking locator, we linked with the long integer runtime library, which has an `l` in its extension. All object modules in a single link must use the same datatype options. There are many other datatype options which are discussed in the *C Compiler* chapter in the *InterTools User's Manual.*

3.4.4 Example 4: Locator Options

This example addresses locator options. We have divided the example into two sections: Separate Data and Locator Commands. We will do the following:

- Locate separate data
- Read locator commands from a file
- Produce a global symbol listing

Separate Data

STEP 1

Generate a source listing for the source file fact.c, which contains an `#pragma separate` preprocessor directive. Write the listing to `fact.sep`. Type:

```
c68020 fact.c -s -l fact.sep
```

Explanation

In STEP 1, we generated `fact.sep`. `fact.sep` is a source listing of `fact.c`, which contains a `#pragma separate` preprocessor directive. The line of the source listing that contains the directive appears below:

```
22 #pragma separate io_port /* another I2 extension
*/
```

`#pragma separate` allows you to control the allocation of variables into segments. You may want to allocate a variable into its own segment to place it at a particular hardware address. If all data after linking is larger than 64K bytes, you must use `#pragma separate` to avoid exceeding the 64K limit on global data. To specify a separate segment in source code, use a `#pragma separate` directive, as shown above.

A common use of `#pragma separate` among embedded system developers is to accommodate memory mapped I/O. Memory mapped I/O refers to hardware built so that reading or writing a particular hardware address causes input or output to an external device. One way to accommodate memory mapped I/O is to declare a separate variable in your source code and locate the segment at the memory mapped address. This use of `#pragma separate` allows C code to manipulate memory mapped I/O without using assembly language routines.

Locator Commands

STEP 2

Compile the two sample C programs and assemble the sample assembly language program for the MC68020 target. Invoke the

linking locator to link and locate, reading locator instructions form the file `loc.cmd`. Write output to the default file. Type:

```
c68020 fact.c circle.c -s
asm68020 fpneg.68k -I inc
llink -i link.lst -L
itools\rtlibs\lib020s\lib\lib020s
      -c loc.cmd -o
form fact.ab
```

Now, generate a global symbol listing for the linked and located file `fact.ab`. Show all symbols in increasing order of memory address. Type:

```
gsmap fact.ab -n -o
```

Explanation

In STEP 2, with the `-c` switch, we directed the linking locator to read locator commands from the file `loc.cmd`. By default, code and data segments are allocated in memory one after another, beginning at address 0. But in a real embedded system, memory areas are often not contiguous. Also, certain address ranges correspond to RAM, ROM or memory–mapped I/O. You must take care to place code and constant data in ROM and read–write data in RAM.

Default conventions may not result in optimal placement. You can control placement of code or data using locator commands. The tutorial files provided with this release include the file `loc.cmd`, a file of commands to locate segments from the input modules. The `-c` switch tells the llink program to read commands from `loc.cmd`. Its contents appear below:

```
--@(#)m/r/tutor/tutor682/loc.cmd        1.4
MEMORY  (#10000);              --Total memory limited to 64K bytes
RESERVE ( #0100 TO #1000);     --Prevent locator from overwriting mem-
RESERVE ( #8000 TO #8100);     --ory reserved for another program
LOCATE (S_io_port : #FF00);    --I/O port's memory location
LOCATE (libcode: AFTER #7000);--put runtime libraries at top of ROM
LOCATE ( {} {code} : #0100);   --Put other code at start of ROM
LOCATE ({data}  : #8100);      --RAM area
```

Before we discuss locator commands, let's discuss the concepts of segment and class. A segment is the smallest piece of code or data that can be independently located in target memory. A collection of segments which share a common attribute define a class. Classes are defined implicitly by the compiler, and may be defined explicitly in assembly language or by using compiler directives and switches. By convention, class names are delimited by curly braces, { }, in locator commands. For instance, the compiler defines a class named {code} which contains all compiler–generated segments containing machine instructions.

The file `loc.cmd` uses the optional `MEMORY` command. The `MEMORY` command defines the true size of target memory. By default the linking locator assumes it can use the entire MC68020 address space.

The `RESERVE` command prevents the location of segments in specified areas. For example, if your system has a ROM–based monitor program, you may wish to avoid loading another program into its address range. Note that in the `TO` syntax, the lower bound is included in the reserve area; the upper bound is not.

The `LOCATE` command actually places code and data in memory areas. You can locate individual segments or whole classes with one command. For example, we located the class {data} after address #8100. The # sign indicates a hexadecimal value.

In the final step, we generated a global symbol listing in increasing address order. The global symbol mapper generates a listing of segments and the definitions of global symbols. It summarizes segment addresses (if you have located the module), length, class, alignment requirements, and combinability. You may require a global symbol table when writing locator commands. Because the symbolic information is derived from an object module, you can generate a global symbol listing either before or after linking and locating. The `gsmap` executable generates a global symbol listing.

It may also prove useful to generate the listing in increasing order of address. With an address order, you can see a sequential listing of segments. The `-n` switch causes gsmap to sort the symbols in increasing order of address. The `-o` switch writes gsmap output to a file by appending the default `.map` suffix. A portion of `fact.map` appears on the following page:

```
Symbol Map for fact.ab                    Sep 6 1997  14:44:08   Page 1

Translator :   llink
Target     :   68020

Global          Address

_FPNEG          00001000 (4096)
_modf           0000102e (4142)
__main          000010f0 (4336)
_getc           00001144 (4420)
_ungetc         000011c2 (4546)
_pow            000011e2 (4578)
_circle         00001370 (4976)
_main           000013b4 (5044)
_factorial      0000147e (5246)
_log            00008100 (33024)
DPEXP           000081e8 (33256)
DPLOG           000082d2 (33490)
DPLN            000082d6 (33494)
DPXTOI          000083e8 (33768)
DPSQRT          00008484 (33924)
        .
        .
        .
Segment         Address          Length       Class      Align  Combine

init@0          00000000 (0)     000008 (8)   <null>     hword  private
S__N5.constant  00000008 (8)     000009 (9)   constant   hword  private
fpneg           00001000 (4096)  00002e (46)  <null>     hword  private
S_modf          0000102e (4142)  0000c2 (194) code       hword  private
init            000010f0 (4336)  000054 (84)  code       hword  private
S_getc          00001144 (4420)  00009e (158) code       hword  private
S_pow           000011e2 (4578)  00018e (398) code       hword  private
S_circle        00001370 (4976)  000044 (68)  code       hword  private
S_main          000013b4 (5044)  0000f2 (242) code       hword  private
        .
        .
        .
Statistics

Segments    :   13
Globals     :   73
Groups      :   1
Code  Size  :   00001888 (6280)
Data  Size  :   00000048 (72)

User Start Address = #10f0
```

3.4.5 Example 5: Formatting Options and Saving Symbol Information

This example addresses formatting options and saving symbol information. The three sections are: Creating Debugging Symbols, Formatting for Multiple ROMs, and Non–Default Formatting. We will do the following:

- Suppress all optimizations that may interfere with debugging.
- Save symbol information for later use by the CrossView Pro debugger.
- Generate a symbol table listing during assembly.
- Use the formatter window and bias options.
- Format using a non–default format.

Creating Debugging Symbols

STEP 1

Compile, assemble and generate a symbol table listing, llink and format. When compiling, suppress all optimizations that may interfere with source–level debugging. At each invocation, save symbol information for CrossView Pro, the InterTools source level cross debugger. Type:

```
c68020 fact.c circle.c -d -do
asm68020 fpneg.68k -d -b -l asm.table
llink -i link.lst -L
itools\rtlibs\lib020s\lib\lib020s -rs idata -x -o
form fact.ab -x -o
```

Explanation

We used the -do switch to suppress optimizations that may interfere with source–level debugging. For more information on optimizations

and their possible effects on debugging, please refer to the *C Compiler* chapter in the *InterTools User's Manual*. We also saved symbol information and prepared for running CrossView Pro, the InterTools source level debugger. At each invocation step, we appended the necessary switch: `-d` for the compiler and assembler to save symbols, `-x` for the linking locator to link in two small debugging routines, and `-x` for the formatter to produce the necessary `.abs` symbol table file.

When we assembled `fpneg.68k`, we generated an assembly listing and a symbol table listing by appending the `-b` switch. We directed output to `asm.table` with the `-l` switch. Portions of `asm.table` appear below:

```
20     18                              |
*  exponent is all 1's; if mantissa is non-zero, then the val is Nan
21     18       2200                   |            MOVE.L D0,D1
22     1A       0281007FFFFF           |            ANDI.L #$007FFFFF,D1
                     ; Check if the value of the mantissa is zero
23     20       66000008               |            BNE FPN000
                     ; Mantissa is non-zero, NaN
24     24                              |
25     24       0A8080000000           |   FPN010   EORI.L #$80000000,D0
                     ; Flip sign bit
26     2A                              |
27     2A       2F00                   |   FPN000   MOVE.L D0,-(SP)
                     ; Push value (negated or not) back on the stack
28     2C       4ED0                   |            JMP  (A0)
; Return
        .
        .
        .
FPN000          type: RELOCATABLE SYMBOL       value : $2A + fpneg
FPN010          type: RELOCATABLE SYMBOL       value : $24 + fpneg
_FPNEG          type: EXTERNAL SYMBOL          value : $0 + fpneg
fpneg           type: RELOCATABLE SECTION      size  : $2E
```

Formatting for Multiple ROMs

STEP 2

Use the formatter bias and window options to create hex files for multiple ROMS each with 16K of memory. Write the output to non–default files. Type:

```
c68020 fact.c circle.c
asm68020 fpneg.68k -I inc
llink -i link.lst -L
itools\rtlibs\lib020s\lib\lib020s
        -rs idata -o
form fact.ab -w 4000 -o fact.1hx
form fact.ab -w 4000 -a 4000 -o fact.2hx
form fact.ab -w 4000 -a 8000 -o fact.3hx
```

Explanation

Suppose your system uses 48K bytes of memory, and you plan to burn your system into three 16K ROM chips, one for each 16K of memory space. You will want to generate three hex files for input to your PROM burner. Each hex file will contain 16K bytes starting at address 0, each extracted in successive 16K byte chunks from the absolute object module (`.ab` file). This example shows how to create those three hex files. We used our small `fact.ab` file, although it does not really contain 48K bytes of code and data.

Here, we use the `-w` and `-a` switches to format three windows of 16K bytes each, starting at hex addresses 0, 4000 and 8000 in succession. The `-w` (windowing) switch chooses the highest address in a hex file after biasing. The `-a` (biasing) switch subtracts a constant hex value from each address in the object module.

In our example, the first formatter invocation creates a hex file that contains hex addresses 0 through 3FFF. The second invocation places the next 16K of addresses, from 4000 to 7FFF, into the second hex file, `fact.2hx`, where 4000 hex has been subtracted from each address. In the third invocation, the formatter creates a hex file that contains the highest 16K of addresses. After these three formatter invocations, we have three hex files (`fact.1hx`, `fact.2hx`, and `fact.3hx`), each of which is suitable for loading into a 16K ROM.

Non–Default Formatting

STEP 3

Format using a non–default format. Type:

```
c68020 fact.c circle.c
asm68020 fpneg.68k
llink -i link.lst -rs idata
      -L itools\rtlibs\lib020s\lib\lib020s -o
      form fact.ab -f bt
```

Explanation

Non–default formatting may be required to meet specific requirements of your target system. In STEP 4 we invoked the formatter to produce a download file in Binary Tekhex format. To specify a non–default format, append the `-f` switch and the desired format. The switch to specify Binary Tekhex is `bt`.

3.5 Introduction to System Building Concepts

These notes are designed to be an extension to the normal Intertools tutorial. They describe things you must consider for system initialization, loading code, and linking C with assembly language in order to build an application.

3.5.1 System Initialization

The InterTools runtime library comes with a set of system initialization templates called either `pmain.68k` or `pmn`*xxx*`.68k`, where the *xxx* refers to the target board (i.e., `pmn332.68k` or `pmn302a.68k`). On the PC, these files are in the `\itools\rtlibs\lib`*xxx*`\src` directory. On Unix hosts, these files are in `/usr/local/itools/rtlibs/lib`*xxx*. When compiling a C module containing the procedure `main`, the compiler automatically generates an unresolved reference to a symbol called `__main` (double underscore). This symbol is defined in all of the `pmain` modules as the start of the initialization routine. Thus, when you link your source modules together, the linker will **AUTOMATICALLY** link in the appropriate `pmain` module in order to satisfy the unresolved reference to `__main`.

The initialization clears registers, sets up the stack, and performs other required power–on initializations such as enabling I/O. The InterTools libraries provide examples which were developed for specific boards. The initialization code must be customized for the actual environment in which it will run. Therefore, it is important that you look at the source code for your particular `pmain`, and change it, if desired. Note that the `pmain` code automatically sets up the first two reset vectors (the `ORG 0` at the bottom), and also sets up `A5` to point to the global data area (the `LEA data,A5` — don't change it if you want A5–relative code !).

Once you have customized the `pmain.68k` module for your particular target board, the next step is to make sure that the modified `pmain` is the one that gets used by your application. There are two ways to accomplish this. The first method is to assemble your modified `pmain.68k` file and then explicitly link it with all of your other source files. This will cause the reference to `__main` to be satisfied (by your `pmain` module), and thus the linker will never look in the library. This is probably the preferable solution if you think that your system initialization routine might undergo further changes. The second method is to add your modified `pmain` to the library, following the procedure in the *Modifying the Libraries* section of the *Runtime Library* appendix in the *InterTools Reference Manual.*

3.5.2 A5–Relative vs. Separate Data Addressing

The Intertools compiler by default uses the A5 register as a pointer to the global data area. This is accomplished via the `LEA data,A5` instruction in `pmain`. Setting up this pointer means that when the compiler manipulates global variables it can now generate instructions using the "address register indirect with offset" addressing mode, which will produce smaller and faster code than the direct addressing mode that the compiler would otherwise have to use. The `data` symbol is set up by the linker to hold the address to be stored in `A5`. For example,

```
i = 2; /* i is a global integer */
```

With A5–relative code:

```
move #2,_i-data(A5) 33fc0002xxxx
```

xxxx is the 16–bit offset from `A5`. On a 68000 this takes 16 cycles plus 3 reads and a write.

With direct addressing:

```
move #2,_i 3b7c0002xxxxxxxx
```

xxxxxxxx is the absolute address of `_i`. On a 68000 this takes 20 cycles plus 4 reads and a write.

The A5–relative instruction allows for a 16–bit offset to be specified, giving the compiler a total global data area of 64K to work with. Any global data that your application has in excess of 64K must be declared as separate using the `#pragma separate` directive (see the *Pragma Separate (Option Separate)* application note for details).

3.5.3 Linking and Locating

The llink step combines three important functions: linking, locating, and ROM processing.

The linking step involves:

1. Telling the linker which modules to link together.
2. Giving the linker access to the libraries it needs to resolve any unresolved references.

The first part is accomplished by listing your `.o1` files (that you got from compiling/assembling your source files) on the command line or in a separate file (using the `-i` switch).

The second part is accomplished by specifying the path to a library index file using the `-L` option. A library index file is essentially a look–up table of symbols and the modules that define those symbols. If the linker finds a symbol that is unresolved in your source files (e.g., `printf`), the linker searches the library index file for the symbol, and upon finding it links in the appropriate library module automatically.

It is important to make sure that you specify the right library index file when linking. If you look in your `\itools\rtlibs\lib000\lib` or `/itools/rtlibs/lib000` (or substitute other target for `lib000`) directory, you will find four distinct library index files: `lib000`, `lib000.l`, `lib000.nf`, and `lib000.lnf`. Library index files with an "`.l`" extension assume that the source files have been compiled with `-L` switch, which changes the default sizes of shorts and ints to 2 and 4 bytes respectively. Library index files with a "`.nf`" extension assume no floating point operations. Thus, if you did not compile with `-L` and used floating point operations, you would use `lib000` as your index file, and so on.

The locating step involves telling the linker where to locate your code and data through the use of a locator command file. The things that you will actually be locating are segments and classes. A segment is a contiguous section of memory containing code or data that has had a name assigned to it either by you or the compiler. A class is a larger classification which contains any number of "member" segments. For example, the Intertools system has created the class `data` to represent all global data which is A5–relative. The class `data` has two member segments, `idata` and `udata`, which contain the initialized and uninitialized global data respectively. The tables which summarize the segments and classes used in Intertools are in the *Linking Locator* chapter of the *InterTools User's Manual*. Below is a table which essentially retranslates this information, but with more emphasis on the segment–to–class relationship:

A:	Segment:	Class:
global variable	`idata` if initialized `udata` if uninitialized	{data}
string literal	`sdata`	{constant}
const qualified variable	`cdata` if you compile with `-cs`, it is treated as a regular global variable otherwise	{constant} if you compile with `-cs`, {data} otherwise
separate variable, no user–defined segment name or class name used	S–variable name	{isep} if initialized variable, {usep} if uninitialized variable
separate variable, with user–defined segment name ONLY	whatever segment name you specified	{separate}

A:	Segment:	Class:
ROM processing segment that you have created	whatever you specified with the -b linker switch	{constant}
C procedure	S_fname, where fname is the first function in the file containing the proc	{code}, unless you change it with the -cc compiler switch
assembly language routine	whatever you specified the segment name to be w/ the SECTION directive. For example, SECTION foo,,"bar" will create a segment foo and a class {bar}	whatever you specified the class to be w/ the SECTION directive, or in the NULL class ({}), if you didn't use the SECTION directive
assembly language that has been absolutely located with ORG	name@address, where name is the SECTION name containing the ORG and address is the absolute address (e.g., init@0)	NULL, although the segment will be located according to the address in the ORG statement regardless of where the null class is located

See the *ORG – Absolute Origin* and *SECTION – Relocatable Program Section* sections of the *Assembler Directives* chapter in the *InterTools Reference Manual* for more information on the ORG and SECTION directives.

Once you know what segments and classes are being created by your compilations, the actual locating process is fairly straight–forward. Within the locator command file, the `LOCATE` directive specifies the segments and/or classes to be located, followed by the absolute address to which they are to be located. Note that class names which are specified in locator command files should be enclosed in curly braces (e.g., {code} and {data}). If you mention more than one segment or class in a particular locate statement they are located in the order you mention them, one right after the other. Segments are considered to be word–aligned, so you must locate segments and classes at even addresses (for separate variables this restriction can be avoided, see the linking and locating example below).

The last thing to consider in the `llink` step is ROM processing. ROM processing is a procedure for storing initialized read–write data in ROM (so that the initial values are not lost when power is off), with the intention of copying this data to RAM at startup (so that the values can be changed). The general procedure works as follows:

1. Establish the identity and contents of a "ROM processing segment". The ROM processing segment is the area of memory in ROM that contains the copies of everything that will be copied to RAM at startup.

   The name of the ROM processing segment is established through the use of the `-b` switch (e.g., `-b _myrompseg`). The catch is that linker symbol names (such as the ROM processing segment name you create with `-b`) are formed by prepending an underscore to the C symbol name. Since you will need to refer to the ROM processing segment's name in C, you must give the ROM processing segment a name that starts with an underscore, so that you can "strip" off the underscore when you refer to it in C.

   The contents of the ROM processing segment are defined via the `-rc` (for classes) and `-rs` (for segments) switches. Any classes or

segments that are specified will be copied into the segment named via the -b switch. In this fashion you can copy any number of segments and classes into the same ROM processing segment.

2. Having established the contents and name of the ROM processing segment, locate it at an appropriate address in ROM. You can do this by locating the segment name explicitly:

```
locate (_myrompseg: #2000);
```

or by the class name constant;

```
locate ({constant}: #2000);
```

3. Locate the segments/classes that you are ROM processing (i.e., the ones you specified with -rs or -rc) at addresses in RAM.

4. Modify your main() routine to call the library routine rcopy. Given that you have used -b to define a ROM processing segment called _myrompseg, the C file containing main() would be modified as follows:

```
#pragma separate myrompseg /* no underscore here, we strip it when
   we're in C */
extern int myrompseg;    /* #pragma separate is necessary to make this
   work */
#include <rcopy.h>
...
...
main()
{
rcopy(&myrompseg);       /* should be the first executable line of
   main() */
...
}
```

Linking/Locating example:

For this example assume a "mythical" 68k target, with ROM located at address `$0` (exception vector table occupies `$0` through `$3ff`) and RAM at address `$10000`. This target also has a bank of 8 LEDs located at address `$9001` (each triggered on a different bit in the byte), and two hex displays at address `$8000` (each displays a hex digit, and both are enabled upon writing a byte to `$8000`). The application program (`main.c`) continuously increments the displayed value after a specified delay; when the display reads `FF` the bank of LEDs flashes on and off. The program also performs the set–up required for ROM processing in the `llink` step.

```
/* begin main.c */

char full = 0xff;
char unused;

#pragma sep_on segment ledbank
char dummy; /* A segment must be word-aligned, so we must add a
  dummy */
char leds; /* variable to put leds at an odd address */
#pragma sep_off

#pragma separate display
char display;

#pragma separate myromp /* dummy variable for ROM processing
  segment */
extern int myromp;

#include <rcopy.h>

void main(void);
void delay(void);

void main(void)
{
                        rcopy(&myromp); /* call to rcopy to do ROM
  processing */
                        leds = display = 0;
                        while(1)
                        {
                                delay();
                                leds = 0;
                                display++;
                                if (display == full) leds = full;
                        }
}

void delay(void)
{
                        return; /* At some point we could insert an
  appropriate delay routine */
}
/* end main.c */
```

Now, if we compile this file (`c68332 main.c -S` *<path to system includes>*), the segments and classes that are created by this compilation are as follows:

- Segment `idata`, which has class {data}, contains the one initialized non–separate global variable in the program (full).
- Segment `udata`, which has class {data}, contains the one uninitialized non–separate global variable in the program (unused).
- Segment `ledbank` contains the separate variables `dummy` and `leds`. Since the `#pragma` directive specifies a segment name (`ledbank`) but no class, the class defaults to {separate}.
- Segment `S_display` contains the variable display and has class {usep}.
- Segment `S_main` contains the code for the routines `main()` and `delay()`, and has class {code}.

Moving on to the `llink` step, we want to accomplish the following:

1. Memory map the variables `leds` and `display` so that they occupy hex addresses `$9001` and `$8000` respectively.
2. ROM process the segment `idata`, locate the ROM processing segment in ROM and reserve space for `idata` (where it will be copied to) in RAM.
3. Locate all code and constant stuff in ROM, and all read–write data in RAM.

This is accomplished by the following `llink` command line:

```
llink main.ol -L <path to index file> -c loc_main.cmd -rs idata -b _myromp -o
```

and locator command file, `loc_main.cmd`:

```
locate({code}{}{constant}: #2000); -- locate code/constant/null
classes in ROM
locate(S_display: #8000); -- map hex displays to $8000
locate(ledbank: #9000); -- satisfies linker's desire for
word-alignment
locate({data}: #10000); -- locate the data (both idata and udata) in
RAM
```

After the `llink`, try running `gsmap` on `main.ab` (`gsmap main.ab -o`) and look at the file `main.map`. In addition to the list of global variables at the beginning, you should also find the following chart of segments:

Segment	Address	Length	Class	Align	Combine
S_display	00008000 (32768)	000001 (1)	usep	hword	private
S_main	00002150 (8528)	00003c (60)	code	hword	private
S_memset	00002000 (8192)	0000ae (174)	code	hword	private
S_rcopy	000020f6 (8438)	00005a (90)	code	hword	private
_myromp	0000218c (8588)	000012 (18)	constant	hword	private
idata	00010000 (65536)	000001 (1)	data	hword	private
init	000020ae (8366)	000048 (72)	code	hword	private
init@0	00000000 (0)	000008 (8)	<null>	byte	private
ledbank	00009000	(36864) 000002 (2)	separate	hword	private
udata	00010002	(65538) 000001 (1)	data	hword	private

Note that all of the segments generated as a result of the compilation of `main.c` are present, and that new segments have appeared as a result of the linking process. The `init` and `init@0` segments were created by the linking in of the module `pmn332.ln`, which contains the default system initialization code. This happened automatically due to the existence of a routine named `main()`. The `init@0` segment represents the data explicitly located at address `0` due to the `ORG 0` at the bottom of the `pmn332.68k` file. The `_myromp` segment is the ROM processing segment that we created by invoking the linker with `-rs idata -b _myromp`. It has class constant by default, a fact that we used in the `loc_main.cmd` file (by locating {constant} instead of _myromp). The `S_memset` and `S_rcopy` segments were generated by the external references to the `rcopy` routine.

3.5.4 Linking C and Assembly

To link C and assembly there really is only one trick and that is: **OBEY THE RUNTIME AND NAMING CONVENTIONS OF THE COMPILER**. Following is a list of important conventions:

- Each symbol referenced in a C program must have an underscore prepended to it when you reference it in assembly language. So, if you declare a variable "`int i;`" in C and then want to access that variable in assembly language, you would do something like `move #2,_i`. Likewise, a function called `main` in C will be called `_main` in assembly.
- The compiler, when calling a function with parameters, will push them onto the stack in reverse order. Also, parameters that are 8 bits in size will be pushed onto the stack as 16 bits (with the high order byte undefined). So, as an example, let's say you have an external function called `foo` that you call in C like so:

```
extern void foo(char,int,long);
...
char c;                    | "e" (low word)
|
int d;        8(A7)->      | "e" (high word)
|
long e;       6(A7)->      | "d"        |
...           4(A7)->      | garbage | "c"
|
foo(c,d,e);                | return addr (low word)
|
                  A7->     | return addr (high word)
|
```

The compiler will push `e` on the stack first, followed by `d` and `c` (which will be pushed as a word), followed by the return address. If `foo` were actually defined in assembly language (`as _foo`), you would expect parameter `c` to be at location `5(a7)` (not `4(a7)`, since the high–order byte is junk), followed by `d` at `6(a7)`, and then `e` at `8(a7)`.

- The compiler will expect integer return values to be placed in `D0`, and pointer return values to be placed in `A0`. For hardware floating point, floats and doubles are returned in `FP0`. Otherwise, floats are returned in `D0` and doubles are returned in a temporary stack location.

- The compiler considers `D0`, `D1`, `A0`, and `A4` (`FP0` and `FP4` also for hardware floating points) to be scratch registers. As such, you never need to worry about saving and restoring these registers in your assembly language programs. `A5`, `A6`, and `A7` are used by the compiler as the pointer to the global data area, the frame pointer, and stack pointer respectively, and you

should not load values into these registers unless you are using a compiler switch which suppresses the default use of these registers (i.e., `-n5` or `-n6`). All other registers (`D2–D7`, `A1–A3`) must be saved and restored if your assembly language routine writes into them. The compiler will be following the same convention, so you **cannot** leave a "live" value in any of the scratch registers prior to making a function call, because the compiler will not be saving and restoring the scratch registers either, and will happily overwrite any useful data that might exist in these registers. Likewise, you **can** leave useful data in any of the preserved registers (`D2–D7,A1–A3`) prior to calling a C function from assembly language, since the compiler will save and restore these registers if it uses them.

Here is an example to illustrate some of the above concepts. Consider a function `foo`, which takes two parameters (`p1` and `p2`, both `int`'s), and returns the value `(2*p1 + 2*p2)`. The function `foo` is to be written in assembly language and it calls another function `bar`, which is written in C. The `bar` function takes an `int` parameter and returns `2*parameter`.

In file `cfile.c`:

```
extern int foo(int,int);
int bar(int);
extern int a, b, c;

void t(void)
{
    c = foo(a,b);
}

int bar(int p1)
{
    return(2*p1);
}
```

In file `asmfile.68k`:

```
                        XDEF _foo ; make _foo public
                        XREF _bar ; "extern" the _bar symbol so you
can use it
                        SECTION fooseg,,"code"
_foo
                        MOVE.L    D2,-(A7)    ; save D2, we'll be using
it
                        MOVE      8(A7),-(A7) ; get value of a, push on
stack for call to bar
                        JSR       _bar    ; call bar(a)
                        ADDQ.L    #2,A7   ; clean up stack (clear off
a)
                        MOVE      D0,D2   ; D0 has return value from
bar(a), save it
* since it is a scratch register and our second call to bar will
clobber it
                        MOVE      10(A7),-(A7)    ; get value of b,
push on stack for call to bar
                        JSR       _bar    ; call bar(b)
                        ADDQ.L    #2,A7   ; clean up stack (clear off
b)
                        ADD       D2,D0   ; D0 has return value from
bar(b) this time,
* we add in the value of bar(a) (still safe and sound in D2, since the
compiler
* will save and restore D2 if bar uses it)
                        MOVE.L    (A7)+,D2    ; restore D2 before
exiting
                        RTS           ; now we can return, since the
return value
* from foo, currently resides in D0, which is what the compiler
expects
```

3.6 Tutorial Conclusion

The examples above address hypothetical but common needs of the embedded system developer. The tutorial addresses basic topics and may be helpful for periodic review and reference. With an understanding of InterTools terminology and practice using the tutorial files, you can approach this manual with fundamental knowledge for detailed applications.

Index

B

bin directory, 3–4

Building an executable, 3–21

C

C++ compiler, files, 3–9

conventions, documentation, 1–5–1–30

customer support, 1–17, 1–22–1–30

D

data addressing, 3–49–3–64

documentation, 1–2–1–30
  conventions, 1–5–1–30

E

EDE, Invoking tools from, 3–15

examples, directory, 3–14

F

formatter, 1–14–1–30

G

global symbol mapper, 1–15–1–30

gsmap. *See* global symbol mapper

H

help, on–line, 1–19–1–30

I

InterTools, toolchain, 1–10

Invoking tools, 3–15

Invoking tools from command line, 3–19

Invoking tools from EDE, 3–15

L

librarian, 1–15–1–30

linking C and assembly, 3–60–3–64

Linking Locator, Rom Processing, 1–13

M

MC68000, 3–6, 3–7

microprocessor family, 1–17

O

on–line help, 1–19–1–30

P

Path, Setting, 3–2

R

RAM, 1–13

rcopy, 1–13

ROM processor, and the linking locator, 1–13

run time libraries, directory, 3–11

S

Setting , path, 3–2

support, customer, 1–17, 1–22–1–30

symbol list utility, 1–16–1–30

symlist. *See* symbol list utility

system building concepts, 3–47–3–64

system initialization, 3–48–3–64

T

Tools, invoking, 3–15